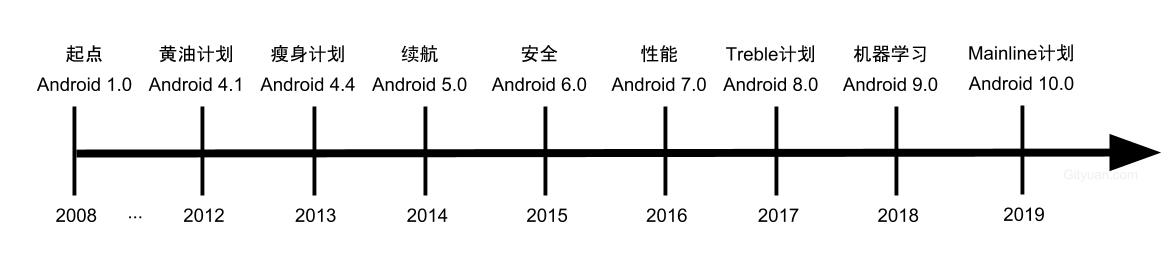
# Android技术架构演进与未

众所周知，Android是谷歌开发的一款基于Linux的开源操作系统，每年迭代一次大版本升级。 小米、华为、OPPO、VIVO、三星等各大厂商对Android原生系统进行二次开发衍生出具有各家特色的系统（比如MIUI），为手机、电视、平板电脑、手表等数十亿设备提供平台支持，使得Android作为全球最受欢迎的移动操作系统。Android诞生至今已有10余年，这一路走来Android遇到哪些问题？大版本升级朝着什么方向演进？Android的未来如何？

## 系统演进



系统演进趋势：每个Android大版本的更新迭代前行，历经10余年，在用户体验、流畅性、续航、安全、隐私、机器学习等方面都取得较大的改进。图中是每个大版本中最具代表性的特征标记在图中，并不代表着该版本全部特征，同样专项计划也不是只在某一个版本执行，比如续航和性能优化，每一个版本都在持续改进中，Treble计划也一直在迭代至今。

* 从Android 1.0发展到Android 4.0，系统各项功能和特性迭代到一个较完善的阶段；
* Android 4.1系统，Google开展了黄油计划（Project Butter），为了让Android系统摆脱UI交互上的严重滞后感，希望能像“黄油”一样顺滑。 核心原理是系统框架中的渲染和动画统一采用垂直同步技术(VSYNC)，以及三重缓冲技术(Triple Buffer)，让滑动、翻页等操作更加一致与顺滑。
* Android 4.4系统，Google开展了瘦身计划（Project Svelte），力求降低安卓系统的内存使用，解决低端机型升级难的问题，让Android 4.4可正常运行在所有Android手机，从而减少安卓系统继续碎片化。UI设计上，支持新的“沉浸式模式”，用户界面由过去的黑色与蓝色为主的色调转向带有透明度的浅色系，视觉语言变得更加明亮与现代化。
* Android 5.0系统，Google开展了伏特计划（Project Volta），力求提升续航能力，这方面Google落后于业界厂商，厂商直面用户对续航尤为迫切，往往系统资源管控更为严格。另外，系统采用全新的ART，抛弃Dalvik虚拟机，大幅提升运行效率。UI设计上，使用全新的扁平化Material Design设计风格，更加清新与质感的设计，统一Android设备的外观和使用体验。
* Android 6.0系统，Google引入新的运行时权限，让用户能够更好地了解和控制权限；引入了Doze模式，进一步提升电池续航能力。UI设计上，新增夜间模式，大幅改进通知栏，让通知更简洁。
* Android 7.0系统，引入新的JIT编译器，对AOT编译器的补充，可节省存储空间和加快更新速度；进一步优化Doze唤醒机制；UI设计上，支持分屏功能；
* Android 8.0系统，Google开展了计划（Project Treble），重新架构Android，将安卓系统框架与Vendor层解耦，力求彻底解决安卓碎片化这一老大难的问题，这是安卓系统架构最大的变化。系统层面加强对后台服务、广播、位置的管控限制。UI设计上，改进通知栏，智能文本选择和自动填充功能。
* Android 9.0系统，引入神经网络API，采用机器学习的思路来预测用户使用习惯来做省电优化，继续强化Treble计划；文件系统(sdcardf/F2FS)持续提升；私有API的限制进一步规范化Android生态，强化隐私和安全，硬件安全性模块以及统一生物识别身份验证界面。 UI设计上，新的手势导航，加强支持刘海屏，UI搜索界面使用到机器学习，AI正在逐步强化Android系统。多相机？？？无线方面，支持多摄像头和wifi室内导航！！！
* Android 10.0系统，Google开展了主线计划（Project Mainline），相关模块（Modules）不允许厂商直接修改，只能由Google应用商店来更新升级，强化用户隐私、系统安全与兼容性。支持脸部生物识别。
* 系统不断演进，但整体架构基本没有改变，如下图所示。



### 发展历程

先来看看Android系统的发展过程，从2008年发布Android 1.0系统，直到2019年即将发布Android 10.0系统，下面列举些重要的时间节点。

* 2003年10月，Andy Rubin团队创办Android公司；
* 2005年8月，谷歌收购Android公司，Andy Rubin担任谷歌工程部副总裁继续负责Android项目；
* 2008年9月，谷歌正式发布Android 1.0系统；
* 2011年1月，Android系统设备的用户总数达到了1.35亿，成为智能手机领域占有量第一的系统；
* 2011年8月，Android手机占据全球智能机市场48%份额，并在亚太地区市场占据统治地位，终结了Symbian系统的霸主地位，跃居全球第一；
* 2012年1月，谷歌Android Market已有10万开发者，推出超过40万应用；
* 2013年11月，Android 4.4正式发布，系统更智能、UI更现代；
* 2013年到2018年，这个阶段安卓进入飞速发展期，被升级的有摄像头、内存、机身、芯片等，原来的3.5寸小屏已退出历史舞台，全面屏、刘海屏、水滴屏已成为当下主流屏幕方案。

## 应用演进

Android系统离不开各App来提供丰富的功能，下面再来简单说一说应用的一些技术演进。

### 移动端跨平台技术

从最开始以Cordova为基础(依赖于WebView)的Hybrid混合开发技术，到React Native的桥接（将JS转为Native）的技术，再到最新的Flutter技术。Flutter是Google发布的全新的移动跨平台UI框架，渲染引擎依靠跨平台的Skia图形库来实现，依赖系统的只有图形绘制相关的接口，可以在最大程度上保证不同平台、不同设备的体验一致性，逻辑处理使用Dart语言，执行效率比JavaScript高。

另外，Google内部正在开发的另一个操作系统Fuchsia的UI layer采用的是Flutter，也就是说Flutter天然可以支持Android、IOS以及未来的Fuchsia。在大前端方向，对于跨平台开发中一直在不断迭代中寻找更好、更优的解决方案，目前来看Flutter还是更有优势。

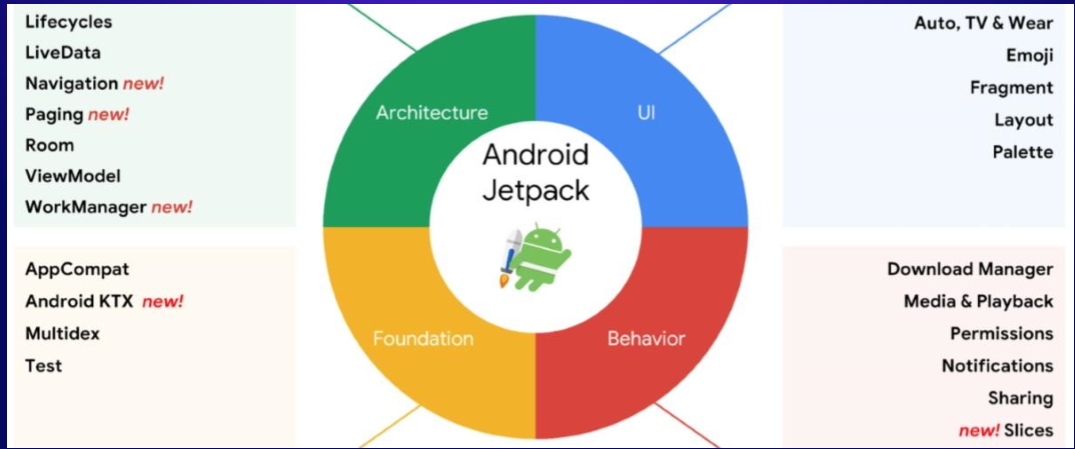
### 热修复与插件化技术

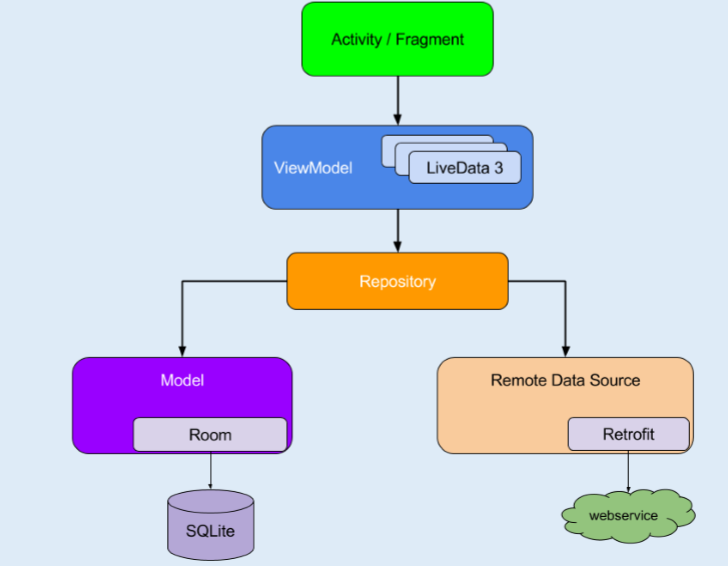
热修复的主要应用场景是为了让用户无感得修复线上缺陷，比如Tinker，Andfix，Sophix等。插件化是为了减少模块耦合，可减少主程序的规模，可按需加载，比如DroidPlugin，OpenAtlas等。关于各个热修复与插件化的细节不再展开，这里就说一点，Android 7.0对Native的NDK的调用限制是手铐，尤其是Android 9.0对Java层SDK的调用限制就是脚铐，那么对于Android应用想再搞插件化之类的黑科技便是带着脚手铐跳舞，能跳但舞姿可能不太美观。没啥前途

### 应用架构

MVC模式（Model–view–controller）但Activity类过于臃肿，为解决这个问题，有了MVP（Model–view–presenter），presenter不仅要操作数据，而且要更新view；再到MVVM（Model-View-ViewModel）解决了MVP大量的手动View和Model同步的问题，提供双向绑定机制

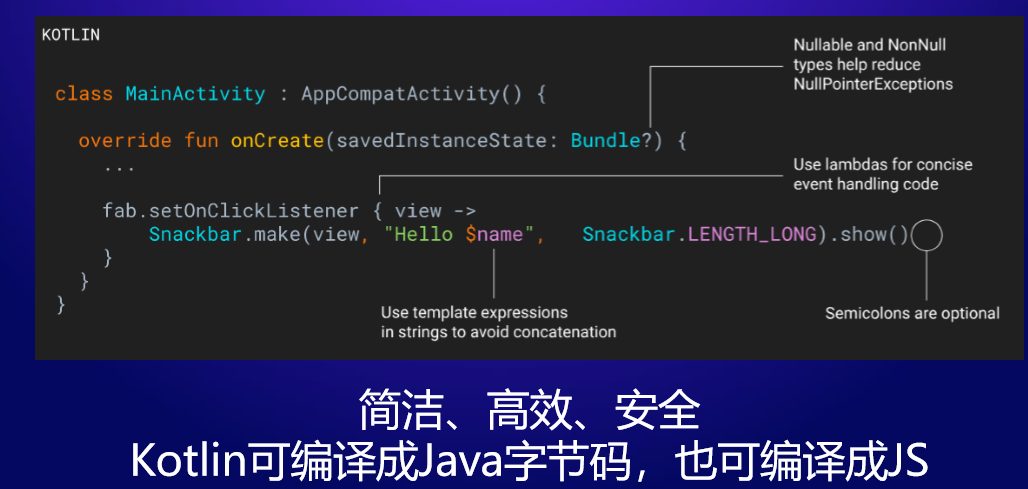
#### Jetpack





### Kotlin

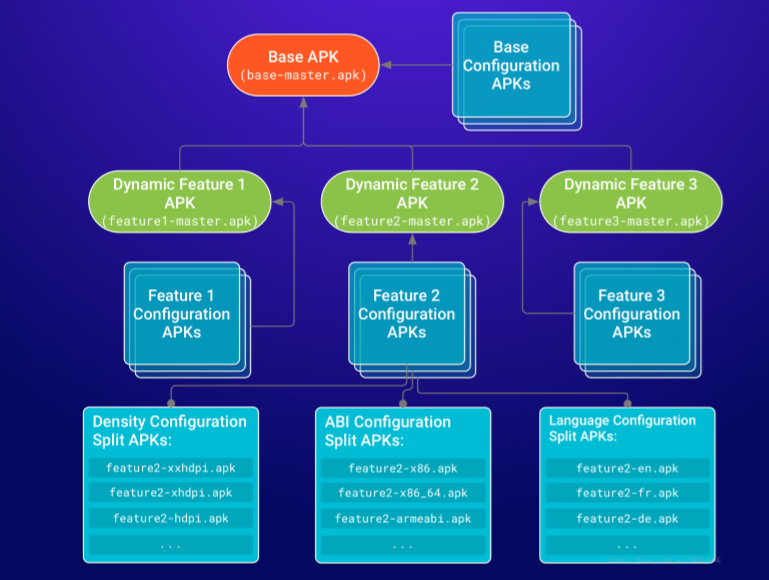
是Google推荐的官方静态编程语言，与Java互通，可相互转换。Kotlin编译成Java字节码，也可以编译成JavaScript，运行在没有JVM的设备上，简洁安全。使用Kotlin更快速地编写Android应用，可以提高开发者的工作效率，少编写样板代码，**被称之为 Android 世界的Swift**。谷歌开发者社区做过一个问卷调查，大概有40%的Android开发者已使用过Kotlin。这里并非鼓励大家一定都要使用Kotlin，学习新语言就像一次投资，要权衡团队成本与收益之间的利弊。效率提升10%而已



### App Bundle

随着应用不断演讲，功能越来越复杂，且应用针对不同屏幕设备、不同国家语言资源都打包在同一个App，导致应用包不断增大，据统计自2012年以来应用包大小增长5倍。虽然现在手机的存储空间越来越大，但用户照片、视频等媒体文件品质在逐渐提升，导致设备可用空间逐渐紧缩。为此Google在去年Google I/O大会讲述Android引入新的App动态化框架（即Android App Bundle，缩写为AAB）。利用Split Apk完成动态加载，使用AAB动态下发方式，可显著缩小应用体积，减少对存储空间的占用。

国内用的很少，应该用起来



#### 好处

#### Size更小

* Google has claimed that it can reduce app sizes up to 50 percent
* Its own apps like YouTube and other apps like LinkedIn which saw a 23% file reduction
* 安装更快
* 支持动态发布

#### 限制

* 仅限于通过 Google Play 发布的应用，（Google进一步巩固自身生态）
* 需要加入到 Google 的 beta program enroll your app in app signing by Google Play in the Play Console
* 最低支持版本Android 5.0 (API level 21)
* 低于Android 5.0 (API level 21) 的版本GooglePlay会优化Size，但不支持动态交付。

#### 成本

* 需要升级到Android Studio 3.2修改工程以便支持App Bundle格式
* 集成Play Core Library

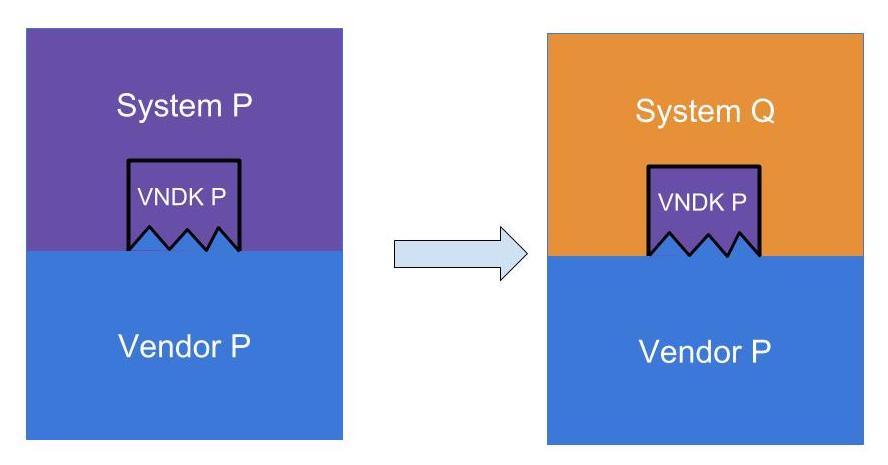
## 未来可能

可能的未来

改善碎片化问题（Treble） • 改进隐私安全与兼容性（Mainline） • 推广图形接口Vulkan • 移动端落地人工智能 • 提升虚拟机 • 统一安卓与物联网的互联互通

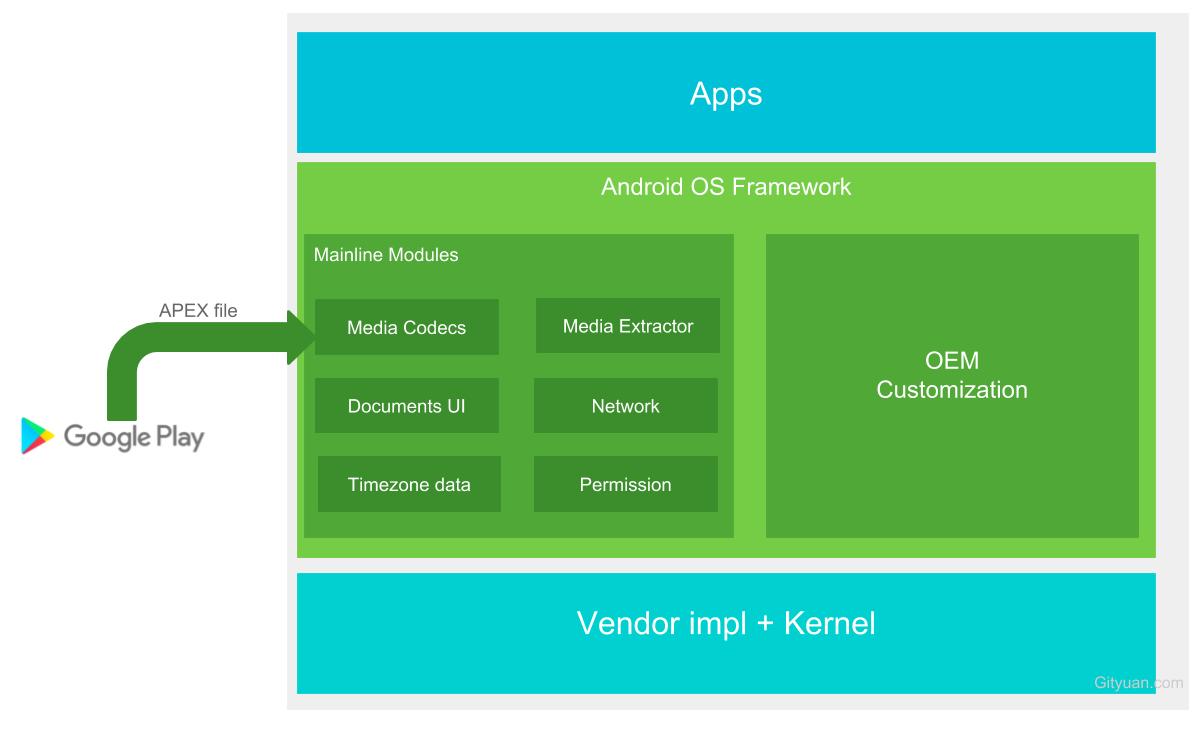
### Treble Project

碎片化

Android历经10余年的迭代，在流畅性、内存、续航、安全、隐私等方面都取得很大的进步，但Android系统的碎片化一直是痛点问题，带来不一致的用户体验。Android的开放性，是其长久发展的主要原因，让大多数的厂商都选择Android系统，但开放性的背后是碎片化，从Android诞生至今问题就一直存在，Google一直在努力从技术角度来解决碎片化问题。从Android 8.0提出Treble项目，重新架构系统将system与vendor解耦合，用于加快Android新版本的适配，效果并不明显，Google继续在后续的Android P以及Android Q一直在不遗余力地持续完善Treble项目，力争加快系统升级速度。如下图，目标是希望在保持Vendor不变的情况下，可以独立升级System模块。  


### Project Mainline

Android系统碎片化，让安全、隐私问题存在风险，且存在体验不一致性问题，但老版本手机的OTA维护升级对厂商来说成本是昂贵的，Ｇoogle感觉到对Android系统掌控力度不足，要想彻底改变，除非不让各大厂商定制化，这势必导致Android手机完全同质化，手机厂商就没法玩了，等于自掘坟墓，Google肯定不会这么干。于是，Google在Android 10.0提出了”Project Mainline“，将对隐私、安全、兼容性造成重大影响的少数模块独立成module，每个module打包成APEX格式（一种类似于APK的新格式），由Google通过应用商店定期来升级，从而保证低版本的手机不会因为碎片化而得不到隐私、安全与兼容性的更新。这些module是由Google维护的主线，各大厂商只能跟Google沟通并将代码upstream到AOSP主线。Google花费了大量的人力在努力完善并推行Mainline，Google希望统一管控的机制，厂商希望最大的自由定制空间，这是一场有趣的角逐，笔者跟团队一起跟Google协商落地module的落地计划，最终将某些module影响较大模块争取Android 11再上线，Mainline更新机制如下图所示。



### Google布局

2014年，发布Android Wear、Android TV 及Android Auto

• 2016年，发布Android Things智能设备系统 •

2016年，开始秘密研发Fuchsia系统

5G + AI的赋能之下，智能汽车、智能家居、 IOT都将会有广阔的市场前景。

#### Fuchsia

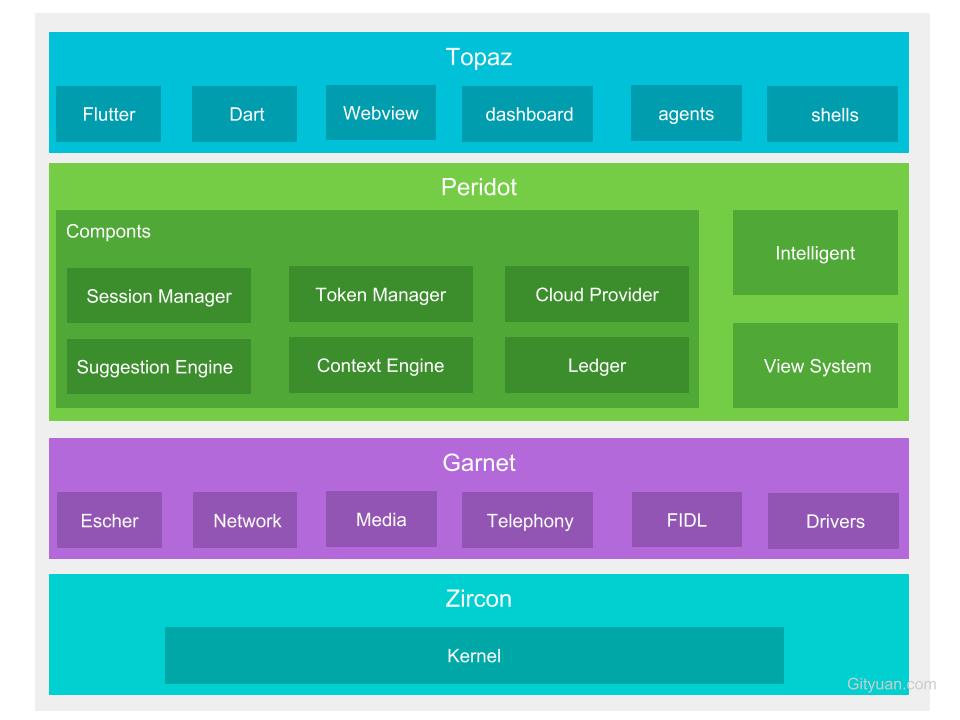
2016年开始，Google有一群超过百人的工程师团队秘密研发一款名为Fuchsia的新系统，该团队很豪华，有来自Android、iPhone、WebOS、Chrome、Flutter等核心工程师，这么多优秀的人在一起研发这个项目，的确值得期待。Fuchsia的内核采用Zircon，UI层采用Flutter框架，底层渲染Escher，支持Vulkan作为底层Graphics API。

提到Vulkan，顺便说一下，去年笔者在美国跟Android团队讨论到并问及Vulkan未来的规划，Google表示未来几年会大力推广Vulkan技术，Vulkan是一种跨平台的高性能低开销的图形接口，在移动设备上比OpenGL ES有着更出色的表现。Vulkan将会是未来Android平台的一个发展方向，尤其是游戏领域，比如王者荣耀Vulkan版本。

在2017年5月，Google的Android工程副总裁Dave Burke称Fuchsia是早期实验项目，而在谷歌内部有很多这类实验项目，存在很大未知变数。2017年11月，Google研发人员表示Fuchsia支持Swift，足见打造统一操作系统的野心。在过去的Google I/O大会只字未提及Fuchsia，Google官方回应不清楚Fuchsia会在出现在什么设备。

Fuchsia会是Android的终结者吗? 笔者认为至少未来五年内不太可能取代Android。当年为了和苹果iOS抗衡，Android系统研发作为Google重中之重，在这种情况下，Android诞生依然花费了Google 3年时间。而Fuchsia只是公司目前的实验项目，且Fuchsia并非基于业界成熟Linux内核，而是采用全新Zircon内核，项目工程路还很远。

笔者大致研究了一下Fuchsia系统源码，总结了一下Fuchsia的整个技术架构图如下。



从Fuchsia技术架构来看，内核层zircon的基础LK是专为嵌入式应用中小型系统设计的内核，代码简洁，适合嵌入式设备和高性能设备，比如IOT、移动可穿戴设备等，目前这些领域还没有标准化级别的垄断者。以及在框架层中有着语音交互、云端以及智能化等模块，由此笔者揣测未来Fuchsia率先应用在音控等智能设备。

Fuchsia基于功能的模块化操作系统，应该会使各组件模块能独立升级更新能力，保证体验一致性。Fuchsia在IOT领域占据一定份额后，加之其良好的跨平台，可以再逐步渗透到移动手机、笔记本电脑等设备，进而三位一体，打造手机、电脑与IOT完美的互联互通的统一平台体验，让多端设备都离不开Fuchsia。在2018年10月，在“蓝牙特别兴趣小组（Bluetooth SIG）”举办的UnPlugFest（UPF）测试大会上，Google再展示了Fuchsia与Android设备的互联性，可以窥见一斑。

Fuchsia的定位更是物联网，再是一统江湖，但路途漫长，至少要５年甚至更远。未来不可知，当然说不定Fuchsia作为实验项目，一直待在实验室，不过这种可能性比较小，做不到一统江湖，在IOT领域发光发亮还是大有可为的。

## 展望未来

追求极致，超预期：打一深进，工具化高效化

勤于思考和总结，提升内功修为：高通，binder问题，不要盲目去解决问题。100个学习，把事情做好

终生学习：大龄程序员！！

#### 操作系统

移动操作系统的演变过程，从按键交互的塞班功能机到触摸屏交互的Android/IOS智能机，从小屏幕手机到全面屏、刘海屏、水滴屏。任何系统无非干两件事：输入和输出，接收到外部输入信号后经过操作系统处理后输出信息。

* 从按键式交互到触屏式交互，伴随着塞班系统到Android系统的转变。未来的交互方式一定会更加生物智能化，当下的触屏交互可以理解成人类的触觉输入方式，未来将朝着人们更常见的听觉输入（语音）和视觉输入（身体姿势、表情等），甚至嗅觉输入（气味变化），也会伴随着新的操作系统的诞生。需要更加无缝地切入生活，而不是“安静，你吵到我的TNT”方式。
* 屏幕从小尺寸到大尺寸，并没有引发操作系统变革，因为技术创新是非连续性，非连续性才会引发第二曲线，诞生新技术。从1960年大型机，到1990年个人笔记本，再到现在的智能手机，设备本身越来越小。未来的设备如果发展非连续变革，可能不再需要实体硬件，随处可输出，一张白纸、一面墙，到那时操作系统的UI架构必然全新的变化。

前面提到Fuchsia系统，笔者认为至少未来五年内不太可能取代Android，但未来可期。新操作系统的崛起源于降维打击，直线超车很难，需要有非连续变革，如果只是某种程度上的改进，很难突破用户习惯、厂商以及生态圈的阻碍。Fuchsia需要降维打击，比如Fuchsia在IOT领域以及新的交互方式都很出色，加上万物无缝式的互联互通的平台，拥有跨平台型特性的Fuchsia有机会成为超级平台。

Android发展至今，已成为全球用户量最广泛的移动操作系统，手机行业竞争异常激烈，经过几番洗牌，国内手机厂商主要是华米OV四大公司，笔者预测在未来五年内国内手机厂商可能只有TOP3，那么Android的未来在哪里呢？

Google在2014年发布Android Wear智能手表系统、Android TV系统以及Android Auto汽车系统，在2016年发布的Android Things智能设备，这些全方位构建安卓的生态圈。在未来在人工智能和5G的赋能下，智能汽车、智能家居、IOT都将会有广阔的市场前景。但就目前人工智能的奇点还没到来，技术还处于前期阶段，一旦奇点来临将会爆炸式发展，或将重新定义生活方式。

汽车的智能化和互联网化是未来一大趋势，Google这两年确实在汽车领域发力，Android Auto在过去一年的用户增长250％。天生的移动特性加上越来越多的互联网服务需求，汽车需要一个具备多种感知能力的系统，或将成为是继手机、电视后Android的下一重点开拓领域。受到驾驶安全的限制，车载场景正好需要将以往的触屏按钮的交互方式，转向语音交互和生物感知，车舱内是天然的语音交互场景，而不再是“安静，你吵到我的TNT”，语音和图像识别、人工智能等技术或许会在车载领域得到更大的发展。

#### 职业发展

随着Android的发展，有些人对Android未来感到茫然，经常收到读者私信问，“前辈，从事Android是不是没有前途，找工作困难，希望能给点建议？”。早在2010年市场上有大量Android招聘，基本懂一点Java基础的就可以搞Android，当时是移动互联网创业的高峰期，号称只要做个App就可以创业。“风来了猪都能找到工作，风停了最先摔死的都是猪”，如果你觉得找工作难，那一定是你在混日子，Android中高级以上的人才一直都非常稀缺。

只要在Android领域深耕，做到极致，努力成为这个方向的专家，有精力再提升工程架构思维，软件工程思想都是相通的，境界会得到提升，即便再学习新东西也会非常快。只要一个领域做到极致，即便Android被淘汰了，换新领域面试官依然会相信你也能做到极致。千万不能用一年的工作能力混十年工作经历，否则你的市场价格连一年都不如，成为工作困难户。

#### Android

Android系统迭代更新10余年，在用户体验、性能、功耗、安全、隐私等方面都取得很大的进步，后续版本会持续在内存、文件系统、虚拟机、图形图像等方向优化。随着Android系统功能越来越多，系统架构中有些模块未来可能会被重构，某些服务大锁制约性能，比如Android 8.0优化过binder大锁让性能显著提升。关于图形方面，Vulkan将会是未来Android平台的一个发展方向，尤其是游戏领域。

人工智能在Android系统上目前效果不太显著，Google未来应该还会持续投入，比如在AI预测用户行为加上相应后台管控策略用于提升手机续航。碎片化仍是当下最主要的问题，碎片化也导致用户隐私、安全和体验一致性方面得不到保障，Google专门成立团队致力于Android Mainline，从Android Q开始规划Mainline，未来的版本都将逐步迭代更新。最后说一点，App兼容性问题比较严重，据Google实验统计Android Q系统在全球Top 1000应用的兼容性不达标率4.3%，而中国Top 1000应用的兼容性不达标率17.6%，可见国内Android生态圈更为严峻得多，Google对隐藏API的限制就是一步长远之棋，短期内导致应用不兼容加剧，长期来看生态圈会逐步健康，最理想的情况就是Android系统大版本升级而App兼容性问题不再有，后续Google应和厂商会加强跟主流应用协作规范应用，搭建良好健康的Android生态圈。

以上是笔者对Android系统及对未来的一些拙见，欢迎业界同仁一起探讨。

## 引用

## 小结

支持多摄像头

WIFI室内导航

插件化和热修复没有意义

大前端，目前来看Flutter还是更有优势

# Android架构

## Android架构

Google提供的4层架构图，是非常经典，但只是如垒砖般的方式，简单地分层，而不足表达Android整个系统的启动过程，环环相扣的连接关系，本文更多的是以进程的视角，以分层的架构来诠释Android系统的全貌。



**图解：** Android系统启动过程由上图从下往上的一个过程：Loader -> Kernel -> Native -> Framework -> App，接来下简要说说每个过程：

### 概述

Android系统非常庞大，底层是采用Linux作为基底，上层采用包含虚拟机的Java层以及Native层，通过系统调用(Syscall)连通系统的内核空间与用户空间。用户空间主要采用C++和Java代码，通过JNI技术打通用户空间的Java层和Native层(C++/C)，从而融为一体

下图是Google提供的一张经典的4层架构图，从下往上，依次分为Linux内核，系统库和Android Runtime，应用框架层，应用程序层这4层架构，每一层都包含大量的子模块或子系统。



这只是如垒砖般地分层，并没有表达Android整个系统的内部架构、运行机理，以及各个模块之间是如何衔接与配合工作的。**为了更深入地掌握Android整个架构思想以及各个模块在Android系统所处的地位与价值，计划以Android系统启动过程为主线，以进程的视角来诠释Android M系统全貌**，全方位的深度剖析各个模块功能，争取各个击破。这样才能犹如庖丁解牛，解决、分析问题则能游刃有余。

### 进程视角

深红色：代表0号进程，是在进入刚进入启动时创建的，内核启动完成后便退出；

浅红色：init/kthreadd/Zygote，这3个进程分别会创建大量的内核守护进程、用户空间守护进程以及应用进程，地位主要创建了大量子进程(注意，此处说的不是子线程)；

深紫色：system server/ media server/ servicemanager，这3个进程并不是用于创建子进程，而是对于整个Android架构，有着非常重要的意义；

深蓝色：内核守护进程、用户空间守护进程以及应用进程，这些都是由“深红色”fork生成的；

浅蓝色：各种系统服务、驱动等相关信息。

### 分层视角

开机过程是从图中最下方Loader开始，经过 -> Kernel -> Native -> Framework，一路直至最上层的App层启动。下面来进一步说明

#### Loader

Boot ROM: 当按下电源开机键，引导芯片代码从预设定处(固化在ROM)开始执行，加载引导程序到RAM；

Boot Loader：是启动Android OS之前的引导程序，主要是检查RAM，初始化硬件参数等功能；

#### Kernel

Kernel层是指Android内核层，到这里才刚刚开始进入Android系统。

启动Kernel的swapper进程(pid=0)：该进程又称为idle进程, 系统初始化过程Kernel由无到有开创的第一个进程, 用于初始化进程管理、内存管理，加载Display,Camera Driver，Binder Driver等相关工作；

启动init进程(1号进程)，是Linux系统的用户空间进程，也就是Native层的进程的鼻祖；

启动kthreadd进程（pid=2）：是Linux系统的内核进程，会创建内核工作线程kworkder，软中断线程ksoftirqd，thermal等内核守护进程。kthreadd进程是所有内核进程的鼻祖。

#### Native

这里的Native层主要包括init孵化来的用户空间的守护进程、HAL层以及开机动画等。启动init进程(pid=1),是Linux系统的用户进程，init进程是所有用户进程的鼻祖。

* init进程会孵化出ueventd、logd、healthd、installd、adbd、lmkd等用户守护进程；
* init进程还启动servicemanager(binder服务管家)、bootanim(开机动画)等重要服务
* init进程孵化出Zygote进程，Zygote进程是Android系统的第一个Java进程(即虚拟机进程)，Zygote是所有Java进程的父进程，Zygote进程本身是由init进程孵化而来的

#### Framework

* Zygote进程，是由init进程通过解析init.rc文件后fork生成的，Zygote进程主要包含：
  + 加载ZygoteInit类，注册Zygote Socket服务端套接字；
  + 加载虚拟机；
  + preloadClasses；
  + preloadResouces。
* System Server进程，是由Zygote进程fork而来，System Server是Zygote孵化的第一个进程，System Server负责启动和管理整个Java framework，包含ActivityManager，PowerManager等服务。
* Media Server进程，是由init进程fork而来，负责启动和管理整个C++ framework，包含AudioFlinger，Camera Service等服务

#### App

此后所有的app启动由System Server调用，然后让Zygote fork，Zygote进程孵化出Home进程，这便是用户看到的桌面App；Zygote进程Browser，Phone等App进程；每个App至少运行在一个进程上；

## 通信基石

无论是Android系统，还是各种Linux衍生系统，各个组件、模块往往运行在各种不同的进程和线程内，这里就必然涉及进程/线程之间的通信。

对于IPC(Inter-Process Communication, 进程间通信)，Linux现有管道、消息队列、共享内存、套接字、信号量、信号这些IPC机制，Android额外还有Binder IPC机制，Android OS中的Zygote进程的IPC采用的是Socket机制，在上层system server、media server以及上层App之间更多的是采用Binder IPC方式来完成跨进程间的通信。

对于Android上层架构中，很多时候是在同一个进程的线程之间需要相互通信，例如同一个进程的主线程与工作线程之间的通信，往往采用的Handler消息机制。

想深入理解Android内核层架构，必须先深入理解Linux现有的IPC机制；对于Android上层架构，则最常用的通信方式是Binder、Socket、Handler，当然也有少量其他的IPC方式，比如杀进程Process.killProcess()采用的是signal方式。下面说说Binder、Socket、Handler：

另外，不同层的通信，api设计也纳入了这类，主要有SysCall，JNI。

### Binder

Binder作为Android系统提供的一种IPC机制，无论从系统开发还是应用开发，都是Android系统中最重要的组成，也是最难理解的一块知识点，想了解[为什么Android要采用Binder作为IPC机制？](https://www.zhihu.com/question/39440766/answer/89210950) 可查看我在知乎上的回答。深入了解Binder机制，最好的方法便是阅读源码，借用Linux鼻祖Linus Torvalds曾说过的一句话：Read The Fucking Source Code。下面简要说说Binder IPC原理。

**Binder IPC原理**

Binder通信采用c/s架构，从组件视角来说，包含Client、Server、ServiceManager以及binder驱动，其中ServiceManager用于管理系统中的各种服务。



* 想进一步了解Binder，可查看[Binder系列—开篇](http://gityuan.com/2015/10/31/binder-prepare/)，Binder系列花费了13篇文章的篇幅，从源码角度出发来，讲述Driver、Native、Framework、App四个层面的整个完整流程。根据有些读者反馈这个系列还是不好理解，这个binder涉及的层次跨度比较大,知识量比较广, 建议大家先知道binder是用于进程间通信,有个大致概念就可以.先去学习系统基本知识,等后面有一定功力再进一步深入研究Binder.

**原理篇**

| **序号** | **文章名** | **概述** |
| --- | --- | --- |
| 0 | [Binder系列—开篇](http://gityuan.com/2015/10/31/binder-prepare/) | Binder概述 |
| 1 | [Binder系列3—启动Service Manager](http://gityuan.com/2015/11/07/binder-start-sm/) | ServiceManager守护进程 注册和查询服务 |
| 2 | [Binder系列4—获取Service Manager](http://gityuan.com/2015/11/08/binder-get-sm/) | 获取代理对象BpServiceManager |
| 3 | [Binder系列5—注册服务(addService)](http://gityuan.com/2015/11/14/binder-add-service/) | 注册Media服务 |
| 4 | [Binder系列6—获取服务(getService)](http://gityuan.com/2015/11/15/binder-get-service/) | 获取Media代理，以及DeathRecipient |
| 5 | [Binder系列7—framework层分析](http://gityuan.com/2015/11/21/binder-framework/) | framework层服务注册和查询，Binder注册 |
| 6 | [理解Binder线程池的管理](http://gityuan.com/2016/10/29/binder-thread-pool/) | Binder的startThreadPool过程 |
| 7 | [彻底理解Android Binder通信架构](http://gityuan.com/2016/09/04/binder-start-service/) | startService为主线 |
| 8 | [Binder系列10—总结](http://gityuan.com/2015/11/28/binder-summary/) | Binder的简单总结 |
| 9 | [Binder IPC的权限控制](http://gityuan.com/2016/03/05/binder-clearCallingIdentity/) | clearCallingIdentity/restoreCallingIdentity |
| 10 | [Binder死亡通知机制之linkToDeath](http://gityuan.com/2016/10/03/binder_linktodeath/) | Binder死亡通知机制 |

**驱动篇:**

|  |  |  |
| --- | --- | --- |
| 1 | [Binder系列1—Binder Driver初探](http://gityuan.com/2015/11/01/binder-driver/) | 驱动open/mmap/ioctl，以及binder结构体 |
| 2 | [Binder系列2—Binder Driver再探](http://gityuan.com/2015/11/02/binder-driver-2/) | Binder通信协议，内存机制 |

**使用篇:**

|  |  |  |
| --- | --- | --- |
| 1 | [Binder系列8—如何使用Binder](http://gityuan.com/2015/11/22/binder-use/) | Native层、Framwrok层自定义Binder服务 |
| 2 | [Binder系列9—如何使用AIDL](http://gityuan.com/2015/11/23/binder-aidl/) | App层自定义Binder服务 |

### Socket

Socket通信方式也是C/S架构，比Binder简单很多。在Android系统中采用Socket通信方式的主要：

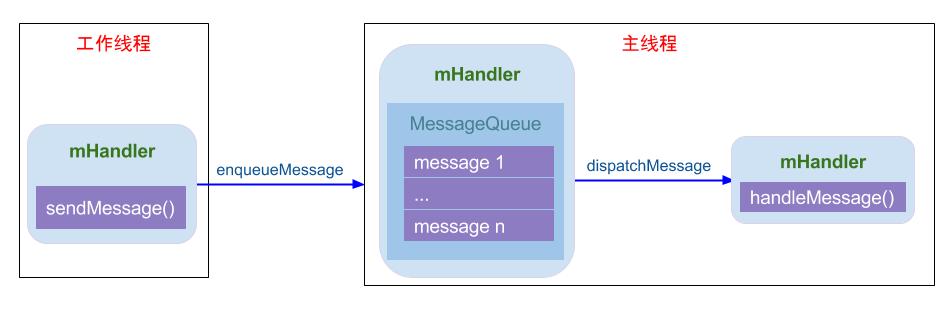
* zygote：用于孵化进程，系统进程system\_server孵化进程时便通过socket向zygote进程发起请求；
* installd：用于安装App的守护进程，上层PackageManagerService很多实现最终都是交给它来完成；
* lmkd：lowmemorykiller的守护进程，Java层的LowMemoryKiller最终都是由lmkd来完成；
* adbd：这个也不用说，用于服务adb；
* logcatd:这个不用说，用于服务logcat；
* vold：即volume Daemon，是存储类的守护进程，用于负责如USB、Sdcard等存储设备的事件处理。

等等还有很多，这里不一一列举，Socket方式更多的用于Android framework层与native层之间的通信。Socket通信方式相对于binder非常简单，所以一直没有写相关文章，为了成一个体系，下次再补上。

### Handler

**Binder/Socket用于进程间通信，而Handler消息机制用于同进程的线程间通信**，Handler消息机制是由一组MessageQueue、Message、Looper、Handler共同组成的，为了方便且称之为Handler消息机制。

有人可能会疑惑，为何Binder/Socket用于进程间通信，能否用于线程间通信呢？答案是肯定，对于两个具有独立地址空间的进程通信都可以，当然也能用于共享内存空间的两个线程间通信，这就好比杀鸡用牛刀。接着可能还有人会疑惑，那handler消息机制能否用于进程间通信？答案是不能，Handler只能用于共享内存地址空间的两个线程间通信，即同进程的两个线程间通信。很多时候，Handler是工作线程向UI主线程发送消息，即App应用中只有主线程能更新UI，其他工作线程往往是完成相应工作后，通过Handler告知主线程需要做出相应地UI更新操作，Handler分发相应的消息给UI主线程去完成，如下图：



由于工作线程与主线程共享地址空间，即Handler实例对象mHandler位于线程间共享的内存堆上，工作线程与主线程都能直接使用该对象，只需要注意多线程的同步问题。工作线程通过mHandler向其成员变量MessageQueue中添加新Message，主线程一直处于loop()方法内，当收到新的Message时按照一定规则分发给相应的handleMessage()方法来处理。所以说，而Handler消息机制用于同进程的线程间通信的核心是线程间共享内存空间，而不同进程拥有不同的地址空间，也就不能用handler来实现进程间通信。

上图只是Handler消息机制的一种处理流程，是不是只能工作线程向UI主线程发消息呢，其实不然，可以是UI线程向工作线程发送消息，也可以是多个工作线程之间通过handler发送消息。更多关于Handler消息机制文章：

* [Android消息机制-Handler(framework篇)](http://gityuan.com/2015/12/26/handler-message-framework/)
* [Android消息机制-Handler(native篇)](http://gityuan.com/2015/12/27/handler-message-native/)
* [Android消息机制3-Handler(实战)](http://gityuan.com/2016/01/01/handler-message-usage/)

要理解framework层源码，掌握这3种基本的进程/线程间通信方式是非常有必要，当然Linux还有不少其他的IPC机制，比如共享内存、信号、信号量，在源码中也有体现，如果想全面彻底地掌握Android系统，还是需要对每一种IPCd机制都有所了解

### API调用（ Syscall && JNI）

这是通信基石之基础

* Native与Kernel之间有一层系统调用(SysCall)层，见[Linux系统调用(Syscall)原理](http://gityuan.com/2016/05/21/syscall/);
* Java层与Native(C/C++)层之间的纽带JNI，见[Android JNI原理分析](http://gityuan.com/2016/05/28/android-jni/)

# android 手机是如何研发出来的

android手机组成：硬件（CPU芯片，触屏，话筒，扬声器，相机，天线，电池，PCB，各种IC等） ＋ 软件：android系统（内核，rom系统，第三方配件驱动等）

oem厂商基于某一平台（高通，联发科，展讯）规划好自己的产品，买来硬件（或者自己生产），组装，烧录系统（根据自己的硬件设备，配件，修改后的aosp，加上第三方驱动）。

现在的像高通等芯片厂商的集成度，越来越高，研制一部android手机的门槛越来越低，但是做一部，高性能，流畅，高度优化，美观，高用户体验的产品，还是需要投入大量资源的。比如基于高通8953平台，研发一部android手机，8953平台就是骁龙625，高通会拿到 aosp源码，针对自己的平台加入相关的驱动，接口代码，厂商购买8953平台，可以拿到 高通修改后的aosp源码，平台说明文档，其他硬件接口文档等。如果厂商想使用莱卡的摄像头，三星的OLED曲面屏幕，这些配件厂商会提供相关的硬件驱动和接口文档，oem厂商集成到系统中，可以进行个性化定制，包括功能，性能，ui等。

# SystemServer

Android体系架构中四种意义上服务:Native服务、Android服务、Init空间的服务、应用层空间的服务。这里的system service属于android服务这一块。

浏览一下Android的system service

Adb shell Service list

从结果看来Android后台有很多的system service,他们是分散在不同进程中的线程实体。

从架构来说，实现了资源文件和代码的解耦：所有的资源文件是framework-res.Apk.提供的，我们的资源包加载也可以这么做啊，包括资源预加载

## 基本概念

### SystemServer

是Android系统的一个核心进程,它是由zygote进程创建的,因此在android的启动过程中位于zygote之后。android的所有服务循环都是建立在 SystemServer之上的。在SystemServer中,将可以看到它建立了android中的大部分服务,并通过ServerManager的add\_service方法把这些服务加入到了ServiceManager的svclist中。从而完成ServcieManager对服务的管理。

### Service Manager

Service manager是管理以上services的一个进程,可以在adb shell中运行ps看看进程列表就知道了。

源代码位于:frameworks/base/cmds/servicemanager

执行方式:

他是用c和c++语言编写的natvie可以执行文件。在Android中称之为EXECUTABLE,这个名称很重要因为Android.mk文件中用这个名字来确定他是可以执行的二进制文件。

### 基本框架

SystemServer的main()函数首先调用的是init1()函数,这是一个native函数,内部会进行一些与Dalvik虚拟机相关的初始化工作。该函数执行完毕后,其内部会调用Java端的init2()函数,这就是为什么Java源码中没有引用init2()的地方,主要的系统服务都是在init2()函数中完成的。

该函数首先创建了一个ServerThread对象,该对象是一个线程,然后直接运行该线程,如以下代码所示:

public static final void init2() { Slog.i(TAG, "Entered the Android system server!");

Thread thr = new ServerThread();

thr.setName("android.server.ServerThread");

thr.start(); }

基本上每个服务都有对应的Java类,从编码规范的角度来看,启动这些服务的模式可归类为三种,如图9-3所示。于是,从ServerThread的run()方法内部开始真正启动各种服务线程。



模式二是指服务类会提供一个getInstance()方法,通过该方法获取该服务对象,这样的好处是保证系统中仅包含一个该服务对象。模式一是指直接使用构造函数构造一个服务,由于大多数服务都对应一个线程,因此,在构造函数内部就会创建一个线程并自动运行。

模式三是指从服务类的main()函数中开始执行。

无论以上何种模式,当创建了服务对象后,有时可能还需要调用该服务类的init()或者systemReady()函数以完成该对象的启动,当然这 些都是服务类内部自定义的。为了区分以上启动的不同,以下采用一种新的方式描述该启动过程。比如当一个服务对象是通过模式一创建,并调用init()完成 该服务的启动,我们就用模式1.2表示;如果构造函数返回后就已经启动,而无须任何其他调用,即什么都不做(nothing),我们就用模式1.1表示。

### 启动服务列表

表9-2列出了SystemServer中所启动的所有服务,以及这些服务的启动模式。

表9-2  SystemServer中启动服务列表

|  |  |  |
| --- | --- | --- |
| 服务类名称 | 作用描述 | 启动模式 |
| EntropyService | 提供伪随机数 | 1.0 |
| PowerManagerService | 电源管理服务 | 1.2/3 |
| ActivityManagerService | 最核心的服务之一,管理 Activity | 自定义 |
| TelephonyRegistry | 通过该服务注册电话模块的事件响应,比如重启、关闭、启动等 | 1.0 |
| PackageManagerService | 程序包管理服务 | 3.3 |
| AccountManagerService | 账户管理服务,是指联系人账户,而不是 Linux 系统的账户 | 1.0 |
| ContentService | ContentProvider 服务,提供跨进程数据交换 | 3.0 |
| BatteryService | 电池管理服务 | 1.0 |
| LightsService | 自然光强度感应传感器服务  // 管理发光二极管和显示背光，我们需要它来显示显示器 | 1.0 |
| VibratorService | 震动器服务 | 1.0 |
| AlarmManagerService | 定时器管理服务,提供定时提醒服务 | 1.0 |
| WindowManagerService | Framework 最核心的服务之一,负责窗口管理 | 3.3 |
| BluetoothService | 蓝牙服务 | 1.0 + |
| DevicePolicyManagerService | 提供一些系统级别的设置及属性 | 1.3 |
| StatusBarManagerService | 状态栏管理服务 | 1.3 |
| ClipboardService | 系统剪切板服务 | 1.0 |
| InputMethodManagerService | 输入法管理服务 | 1.0 |
| NetStatService | 网络状态服务 | 1.0 |
| NetworkManagementService | 网络管理服务 | NMS.create() |
| ConnectivityService | 网络连接管理服务 | 2.3 |
| ThrottleService | 暂不清楚其作用 | 1.3 |

(续表)

|  |  |  |
| --- | --- | --- |
| 服务类名称 | 作用描述 | 启动模式 |
| AccessibilityManagerService | 辅助管理程序截获所有的用户输入,并根据这  些输入给用户一些额外的反馈,起到辅助的效果 | 1.0 |
| MountService | 挂载服务,可通过该服务调用 Linux 层面的 mount 程序 | 1.0 |
| NotificationManagerService | 通知栏管理服务, Android 中的通知栏和状  态栏在一起,只是界面上前者在左边,后者在右边 | 1.3 |
| DeviceStorageMonitorService | 磁盘空间状态检测服务 | 1.0 |
| LocationManagerService | 地理位置服务 | 1.3 |
| SearchManagerService | 搜索管理服务 | 1.0 |
| DropBoxManagerService | 通过该服务访问 Linux 层面的 Dropbox 程序 | 1.0 |
| WallpaperManagerService | 墙纸管理服务,墙纸不等同于桌面背景,  在 View 系统内部,墙纸可以作为任何窗口的背景 | 1.3 |
| AudioService | 音频管理服务 | 1.0 |
| BackupManagerService | 系统备份服务 | 1.0 |
| AppWidgetService | Widget 服务 | 1.3 |
| RecognitionManagerService | 身份识别服务 | 1.3 |
| DiskStatsService | 磁盘统计服务 | 1.0 |

这些服务在又具体定义

@StringDef({

POWER\_SERVICE,

WINDOW\_SERVICE,

LAYOUT\_INFLATER\_SERVICE,

ACCOUNT\_SERVICE,

ACTIVITY\_SERVICE,

ALARM\_SERVICE,

NOTIFICATION\_SERVICE,

ACCESSIBILITY\_SERVICE,

CAPTIONING\_SERVICE,

KEYGUARD\_SERVICE,

LOCATION\_SERVICE,

//@hide: COUNTRY\_DETECTOR,

SEARCH\_SERVICE,

SENSOR\_SERVICE,

STORAGE\_SERVICE,

WALLPAPER\_SERVICE,

VIBRATOR\_SERVICE,

//@hide: STATUS\_BAR\_SERVICE,

CONNECTIVITY\_SERVICE,

//@hide: UPDATE\_LOCK\_SERVICE,

//@hide: NETWORKMANAGEMENT\_SERVICE,

NETWORK\_STATS\_SERVICE,

//@hide: NETWORK\_POLICY\_SERVICE,

WIFI\_SERVICE,

WIFI\_NAN\_SERVICE,

WIFI\_P2P\_SERVICE,

WIFI\_SCANNING\_SERVICE,

//@hide: WIFI\_RTT\_SERVICE,

//@hide: ETHERNET\_SERVICE,

WIFI\_RTT\_SERVICE,

NSD\_SERVICE,

AUDIO\_SERVICE,

FINGERPRINT\_SERVICE,

MEDIA\_ROUTER\_SERVICE,

TELEPHONY\_SERVICE,

TELEPHONY\_SUBSCRIPTION\_SERVICE,

CARRIER\_CONFIG\_SERVICE,

TELECOM\_SERVICE,

CLIPBOARD\_SERVICE,

INPUT\_METHOD\_SERVICE,

TEXT\_SERVICES\_MANAGER\_SERVICE,

APPWIDGET\_SERVICE,

//@hide: VOICE\_INTERACTION\_MANAGER\_SERVICE,

//@hide: BACKUP\_SERVICE,

DROPBOX\_SERVICE,

//@hide: DEVICE\_IDLE\_CONTROLLER,

DEVICE\_POLICY\_SERVICE,

UI\_MODE\_SERVICE,

DOWNLOAD\_SERVICE,

NFC\_SERVICE,

BLUETOOTH\_SERVICE,

//@hide: SIP\_SERVICE,

USB\_SERVICE,

LAUNCHER\_APPS\_SERVICE,

//@hide: SERIAL\_SERVICE,

//@hide: HDMI\_CONTROL\_SERVICE,

INPUT\_SERVICE,

DISPLAY\_SERVICE,

USER\_SERVICE,

RESTRICTIONS\_SERVICE,

APP\_OPS\_SERVICE,

CAMERA\_SERVICE,

PRINT\_SERVICE,

CONSUMER\_IR\_SERVICE,

//@hide: TRUST\_SERVICE,

TV\_INPUT\_SERVICE,

//@hide: NETWORK\_SCORE\_SERVICE,

USAGE\_STATS\_SERVICE,

MEDIA\_SESSION\_SERVICE,

BATTERY\_SERVICE,

JOB\_SCHEDULER\_SERVICE,

//@hide: PERSISTENT\_DATA\_BLOCK\_SERVICE,

MEDIA\_PROJECTION\_SERVICE,

MIDI\_SERVICE,

RADIO\_SERVICE,

HARDWARE\_PROPERTIES\_SERVICE,

//@hide: SOUND\_TRIGGER\_SERVICE,

SHORTCUT\_SERVICE,

//@hide: CONTEXTHUB\_SERVICE,

})

@Retention(RetentionPolicy.SOURCE)

public @interface ServiceName {}

public abstract Object getSystemService(@ServiceName @NonNull String name);

## 启动流程

### 时序



### SystemServer.main

SystemServer.java

|  |
| --- |
| public final class SystemServer {  ...  public static void main(String[] args) {  //先初始化SystemServer对象，再调用对象的run()方法  new SystemServer().run();  }  } |

### SystemServer.run

SystemServer.java

|  |
| --- |
| private void run() {  // Prepare the main looper thread (this thread). 初始化looper，是不是没想到looper会被用在此处？  android.os.Process.setThreadPriority(  android.os.Process.THREAD\_PRIORITY\_FOREGROUND);  android.os.Process.setCanSelfBackground(false);  Looper.prepareMainLooper()  if (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  Slog.w(TAG, "System clock is before 1970; setting to 1970.");  SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  }  ...  if (!SystemProperties.get("persist.sys.language").isEmpty()) {  设置系统的的语言  final String languageTag = Locale.getDefault().toLanguageTag();  SystemProperties.set("persist.sys.locale", languageTag);  SystemProperties.set("persist.sys.language", "");  SystemProperties.set("persist.sys.country", "");  SystemProperties.set("persist.sys.localevar", "");  }  Slog.i(TAG, "Entered the Android system server!");  EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_SYSTEM\_RUN, SystemClock.uptimeMillis());  Looper.prepareMainLooper();// 准备主线程looper  //加载android\_servers.so库，该库包含的源码在frameworks/base/services/目录下  System.loadLibrary("android\_servers");  //检测上次关机过程是否失败，该方法可能不会返回[见小节3.6.1]  performPendingShutdown();  createSystemContext(); //初始化系统上下文  //创建系统服务管理  mSystemServiceManager = new SystemServiceManager(mSystemContext);  LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);  //启动各种系统服务[见小节3.7]  try {  startBootstrapServices(); // 启动引导服务  startCoreServices(); // 启动核心服务  startOtherServices(); // 启动其他服务[见小节4.6]  } catch (Throwable ex) {  Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting system services", ex);  throw ex;  }  //一直循环执行  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

#### 准备SystemServer运行环境

// 准备SystemServer运行环境:设置线程优先级，创建主线层Looper，ActivityThread和SystemContext

#### 设置时间

Sda

*// If a device's clock is before 1970 (before 0), a lot of  
// APIs crash dealing with negative numbers, notably  
// java.io.File#setLastModified, so instead we fake it and  
// hope that time from cell towers or NTP fixes it shortly.***if** (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  
 Slog.w(TAG, **"System clock is before 1970; setting to 1970."**);  
 SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  
}

如何修改Android系统默认时间

https://blog.csdn.net/ljx646566715/article/details/81189914

#### Shiqi

SystemProperties.set(**"persist.sys.timezone"**, **"GMT"**);

#### 设置系统的语言

/1、设置系统的的语言

if (!SystemProperties.get("persist.sys.language").isEmpty()) {

final String languageTag = Locale.getDefault().toLanguageTag();

SystemProperties.set("persist.sys.locale", languageTag);

SystemProperties.set("persist.sys.language", "");

SystemProperties.set("persist.sys.country", "");

SystemProperties.set("persist.sys.localevar", "");

}

这里通过SystemProperties类去读取系统的属性。这个语句的逻辑也比较简单，主要是去设置系统语言的环境。这些属性最后都会被init进程在它们对应的动作列表或者是服务列表中检测到，并且调用相对应的函数去执行。

#### 进程性能统计

系统中很多进程都需要通过SamplingProfilerIntegration去统计性能。

// 2、Enable the sampling profiler.进程性能统计

if (SamplingProfilerIntegration.isEnabled()) {

SamplingProfilerIntegration.start();

mProfilerSnapshotTimer = new Timer();

mProfilerSnapshotTimer.schedule(new TimerTask() {

@Override

public void run() {

SamplingProfilerIntegration.writeSnapshot("system\_server", null);

}

}, SNAPSHOT\_INTERVAL, SNAPSHOT\_INTERVAL);

}

#### 置虚拟机运行内存

//3、 Mmmmmm... more memory!设置虚拟机运行内存

VMRuntime.getRuntime().clearGrowthLimit();

VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);

Build.ensureFingerprintProperty();

Environment.setUserRequired(true);

Bundle.setShouldDefuse(true);

#### 设置环境变量

*// Within the system server, it is an error to access Environment paths without  
// explicitly specifying a user.*Environment.setUserRequired(**true**);

### 服务启动

这三类：boot服务、core服务、其他服务.启动critical级别的服务，他们有复杂的依赖关系，顺序不能乱,自己想加服

#### 载入nativie 服务库

*// Initialize native services.*System.loadLibrary(**"android\_servers"**);

手动加载的啊，也没有什么神奇的

#### createSystemContext()

系统Context初始化过程

**private void** createSystemContext() {  
 ActivityThread activityThread = ActivityThread.systemMain();  
 mSystemContext = activityThread.getSystemContext();  
 mSystemContext.setTheme(DEFAULT\_SYSTEM\_THEME);  
  
 **final** Context systemUiContext = activityThread.getSystemUiContext();  
 systemUiContext.setTheme(DEFAULT\_SYSTEM\_THEME);  
}

android.ddm.DdmHandleAppName.setAppName("system\_process",

UserHandle.myUserId());

**系统APK(framework-res.apk)对应的LoadedApk初始化**

/\*\*

\* Create information about the system package.

\* system package : framework-res.apk,packagename为android

\* Must call {@link #installSystemApplicationInfo} later.

\*/

LoadedApk(ActivityThread activityThread) {

mActivityThread = activityThread;

mApplicationInfo = new ApplicationInfo();

**// packageName为"android",这个APK为framework-res.apk**

**mApplicationInfo.packageName = "android";**

**mPackageName = "android";**

mAppDir = null;

mResDir = null;

mSplitAppDirs = null;

mSplitResDirs = null;

mOverlayDirs = null;

mSharedLibraries = null;

mDataDir = null;

mDataDirFile = null;

mLibDir = null;

mBaseClassLoader = null;

mSecurityViolation = false;

mIncludeCode = true;

mRegisterPackage = false;

mClassLoader = ClassLoader.getSystemClassLoader();

mResources = Resources.getSystem();

}

SystemServer.createSystemContext()分析到这里就基本结束了，从这里可以看到，SystemServer.createSystemContext()主要是创建了ActivityThread，获取了Context，这个Context是系统进程（framework-res.apk）运行的环境。另外，从代码中也可以看到:  
1)ActivityThread中用了一个集合来保存Application，说明一个进程里面可以包含多个Application;  
具体实现：在APK的AndroidManifext.xml中配置<manifest>标签的android:sharedUserId和配置<application>标签的android:process相同即可，这样配置之后，这些APK就能够运行在同一个进程，实现数据共享；  
2)Application是在LoadedApk中通过Intrumentation创建出来的(LoadedApk.makeApplication()-->Instrumentation.newApplication()-->Application.attach())，Application通过attach()方法持有LoadedApk对象，Application和LoadedApk是一一对应的关系。

<https://www.jianshu.com/p/7d4b605f5060>

这里的context是系统的，没有注册服务缓存。因此，获取服务需要：

mWifiManager = IWifiManager.Stub.asInterface(

ServiceManager.getService(Context.WIFI\_SERVICE));

而不是：

// WifiManager wifiManager = (WifiManager) context.getSystemService(Context.WIFI\_SERVICE);

#### startBootstrapServices

|  |
| --- |
| public final class SystemServer {  private void startBootstrapServices() {  ...  //phase100  mSystemServiceManager.startBootPhase(SystemService.PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY);  ...  //启动Installer    //启动ActivityManagerService    //启动PowerManagerService    //启动DisplayManagerService    //启动PackageManagerService    }  } |

//Installer类，该类是系统安装apk时的一个服务类，该类是系统安装apk时的一个服务类

Installer installer = mSystemServiceManager.startService(Installer.class);

##### ActivityMS

Sda

*// Activity manager runs the show.*traceBeginAndSlog(**"StartActivityManager"**);  
mActivityManagerService = mSystemServiceManager.startService(  
 ActivityManagerService.Lifecycle.**class**).getService();  
mActivityManagerService.setSystemServiceManager(mSystemServiceManager);  
mActivityManagerService.setInstaller(installer);  
traceEnd();

##### setSystemProcess

**public void** setSystemProcess() {  
 **try** {  
 ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(  
 **"android"**, STOCK\_PM\_FLAGS | MATCH\_SYSTEM\_ONLY);  
 mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());  
  
 **synchronized** (**this**) {  
 ProcessRecord app = newProcessRecordLocked(info, info.processName, **false**, 0);  
 app.persistent = **true**;  
 app.pid = MY\_PID;  
 app.maxAdj = ProcessList.SYSTEM\_ADJ;  
 app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);  
 **synchronized** (mPidsSelfLocked) {  
 mPidsSelfLocked.put(app.pid, app);  
 }  
 updateLruProcessLocked(app, **false**, **null**);  
 updateOomAdjLocked();  
 }  
 } **catch** (PackageManager.NameNotFoundException e) {  
 **throw new** RuntimeException(  
 **"Unable to find android system package"**, e);  
 }  
}

但system\_server的ADJ并非等于-900，而是-800？是由于startPersistentApps()过程直接把其adj重新被设置为-800，这算是一个小BUG，但 其实目前来说对于ADJ<0的进程，LMK不会杀，两者没有什么区别。

#### startCoreServices

/\* 启动一些关键服务\*

//启动LightsService

//启动BatteryService

//启动UsageStatsService

//启动WebViewUpdateService

#### startOtherServices

*// For debugging RescueParty***if** (Build.IS\_DEBUGGABLE && SystemProperties.getBoolean(**"debug.crash\_system"**, **false**)) {  
 **throw new** RuntimeException();  
}

//启动AccountManagerService 帐号管理

//启动ContentService 内容管理

//启动SystemProviders

//启动VibratorService 震动

//启动ConsumerIrService 远程控制周边设备

//启动AlarmManagerService 闹钟

//启动Watchdog

//启动WindowManagerService 窗口管理

//启动InputMethodManagerService 输入法

//截获用户输入，给一些额外反馈，view的点击、焦点事件分发

//启动AccessibilityManagerService

//启动MountService 磁盘加载服务

//启动LockSettingsService 锁屏、手势

//启动PersistentDataBlockService

//启动DevicePolicyManagerService //保证和API8兼容

//启动StatusBarManagerService 状态栏

//启动ClipboardService 剪切板

//启动NetworkManagementService 网络管理

//启动NetworkScoreService

//启动NetworkStatsService 网络统计

//启动NetworkPolicyManagerService 维护网络使用策略

//启动WifiService WIFI服务

//启动ConnectivityService 网络连接状态

//启动网络发现服务

//启动TextServicesManagerService 文本服务，如文本检查

//启动UpdateLockService

//启动LocationManagerService 位置

//启动**CountryDetectorService** 国家检测

//启动SearchManagerService 搜索

//启动DropBoxManagerService

//启动WallpaperManagerService 墙纸

//启动AudioService 音频

//启动USB服务

//WiredAccessoryManager 有线接入，耳机之类的

//启动SerialService 不知道是啥

//启动TwilightService 指明用户当前位置是否为晚上，配合调整夜间模式

//启动UiModeManagerService 界面模式

//启动JobSchedulerService 任务调度

//启动BackupManagerService 备份

//启动APPWIDGET\_SERVICE 窗口小部件管理

//启动VOICE\_RECOGNITION\_MANAGER\_SERVICE 声音重置服务

//启动DiskStatsService 硬盘统计服务

//启动NetworkTimeUpdateService 网络时间更新服务

//启动CommonTimeManagementService 时间管理服务

//启动DreamManagerService 屏幕保护

//负责将预加载的bitmap组装成纹理贴图，生成的纹理贴图可以被用来跨进程使用，以减少内存

//启动AssetAtlasService

//启动打印服务

//RestrictionsManagerService ？

//MediaSessionService ？

//HdmiControlService HDMI服务

//TvInputManagerService TV输入管理

//MediaRouterService ？

//TrustManagerService 信任管理

//FingerprintService 指纹

//BackgroundDexOptService ？

//LauncherAppsService

//MediaProjectionManagerService

//如果“安全” ，禁用JIT编译模式，不安全则开启JIT 何为安全？？？？

// VMRuntime.getRuntime().disableJitCompilation()

//MmsServiceBroker ？

//接下来启动各种APP，省略……

/\*

启动各种各样的“杂项”服务

有一个关键标志变量 disableNonCoreServices 是否把非核心的服务禁用——————如果是，下面很多服务就不会启动了

Starts a miscellaneous grab bag of stuff that has yet to be refactored

\* and organized.

private void startOtherServices() {

...

//phase480 和phase500

mSystemServiceManager.startBootPhase(SystemService.PHASE\_LOCK\_SETTINGS\_READY);

mSystemServiceManager.startBootPhase(SystemService.PHASE\_SYSTEM\_SERVICES\_READY);

...

//[见小节4.7]

mActivityManagerService.systemReady(new Runnable() {

@Override

public void run() {

//phase550

mSystemServiceManager.startBootPhase(

SystemService.PHASE\_ACTIVITY\_MANAGER\_READY);

...

//phase600

mSystemServiceManager.startBootPhase(

SystemService.PHASE\_THIRD\_PARTY\_APPS\_CAN\_START);

}

}

}

* start: 创建AMS, PMS, LightsService, DMS.
* phase100: 进入Phase100, 创建PKMS, WMS, IMS, DBMS, LockSettingsService, JobSchedulerService, MmsService等服务;
* phase480 && 500: 进入Phase480, 调用WMS, PMS, PKMS, DisplayManagerService这4个服务的systemReady();
* Phase550: 进入phase550, 执行AMS.systemReady(), 启动SystemUI, WebViewFactory, Watchdog.
* Phase600: 进入phase600, 执行AMS.systemReady(), 执行各种服务的systemRunning().
* Phase1000: 进入1000, 执行finishBooting, 启动启动on-hold进程.

##### SystemService.PHASE\_LOCK\_SETTINGS\_READY

##### WallpaperManagerService

config\_enableWallpaperService的config值

*<!-- Component name of the built in wallpaper used to display bitmap wallpapers. This must not be null. -->*<**string name="image\_wallpaper\_component" translatable="false"**>com.android.systemui/com.android.systemui.ImageWallpaper</**string**>

mImageWallpaper = ComponentName.unflattenFromString(  
 context.getResources().getString(R.string.image\_wallpaper\_component));

##### startSystemUi

//启动系统UI

static final void startSystemUi(Context context) {

Intent intent = new Intent();

intent.setComponent(new ComponentName("com.android.systemui",

"com.android.systemui.SystemUIService"));

intent.addFlags(Intent.FLAG\_DEBUG\_TRIAGED\_MISSING);

//Slog.d(TAG, "Starting service: " + intent);

context.startServiceAsUser(intent, UserHandle.SYSTEM);

}

## App如何启动

对于普通的app进程,跟system\_server进程的启动过来有些类似.不同的是app进程是向发消息给system\_server进程, 由system\_server向zygote发出创建进程的请求.

[理解Android进程创建流程](http://gityuan.com/2016/03/26/app-process-create/), 可知进程创建后 接下来会进入ActivityThread.main()过程。

### ActivityThread.main

|  |
| --- |
| public static void main(String[] args) {  ...  Environment.initForCurrentUser();  ...  Process.setArgV0("<pre-initialized>");  //创建主线程looper  Looper.prepareMainLooper();  ActivityThread thread = new ActivityThread();  thread.attach(false); //attach到系统进程  if (sMainThreadHandler == null) {  sMainThreadHandler = thread.getHandler();  }    //主线程进入循环状态  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

app进程的主线程调用栈的栈底如下:

|  |
| --- |
| ...  at android.app.ActivityThread.main(ActivityThread.java:5442)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

跟前面介绍的system\_server进程调用栈对比:

|  |
| --- |
| at com.android.server.SystemServer.main(SystemServer.java:175)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

### main

SystemServer.java

|  |
| --- |
| public final class SystemServer {  ...  public static void main(String[] args) {  //先初始化SystemServer对象，再调用对象的run()方法  new SystemServer().run();  }  } |

### SystemServer.run

SystemServer.java

|  |
| --- |
| private void run() {  if (System.currentTimeMillis() < EARLIEST\_SUPPORTED\_TIME) {  Slog.w(TAG, "System clock is before 1970; setting to 1970.");  SystemClock.setCurrentTimeMillis(EARLIEST\_SUPPORTED\_TIME);  }  ...  Slog.i(TAG, "Entered the Android system server!");  EventLog.writeEvent(EventLogTags.BOOT\_PROGRESS\_SYSTEM\_RUN, SystemClock.uptimeMillis());  Looper.prepareMainLooper();// 准备主线程looper  //加载android\_servers.so库，该库包含的源码在frameworks/base/services/目录下  System.loadLibrary("android\_servers");  //检测上次关机过程是否失败，该方法可能不会返回[见小节3.6.1]  performPendingShutdown();  createSystemContext(); //初始化系统上下文  //创建系统服务管理  mSystemServiceManager = new SystemServiceManager(mSystemContext);  LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);  //启动各种系统服务[见小节3.7]  try {  startBootstrapServices(); // 启动引导服务  startCoreServices(); // 启动核心服务  startOtherServices(); // 启动其他服务[见小节4.6]  } catch (Throwable ex) {  Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting system services", ex);  throw ex;  }  //一直循环执行  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

### 服务启动

|  |
| --- |
| public final class SystemServer {  private void startBootstrapServices() {  ...  //phase100  mSystemServiceManager.startBootPhase(SystemService.PHASE\_WAIT\_FOR\_DEFAULT\_DISPLAY);  ...  }  private void startOtherServices() {  ...  //phase480 和phase500  mSystemServiceManager.startBootPhase(SystemService.PHASE\_LOCK\_SETTINGS\_READY);  mSystemServiceManager.startBootPhase(SystemService.PHASE\_SYSTEM\_SERVICES\_READY);  ...  //[见小节4.7]  mActivityManagerService.systemReady(new Runnable() {  @Override  public void run() {  //phase550  mSystemServiceManager.startBootPhase(  SystemService.PHASE\_ACTIVITY\_MANAGER\_READY);  ...  //phase600  mSystemServiceManager.startBootPhase(  SystemService.PHASE\_THIRD\_PARTY\_APPS\_CAN\_START);  }  }  }  } |

* start: 创建AMS, PMS, LightsService, DMS.
* phase100: 进入Phase100, 创建PKMS, WMS, IMS, DBMS, LockSettingsService, JobSchedulerService, MmsService等服务;
* phase480 && 500: 进入Phase480, 调用WMS, PMS, PKMS, DisplayManagerService这4个服务的systemReady();
* Phase550: 进入phase550, 执行AMS.systemReady(), 启动SystemUI, WebViewFactory, Watchdog.
* Phase600: 进入phase600, 执行AMS.systemReady(), 执行各种服务的systemRunning().
* Phase1000: 进入1000, 执行finishBooting, 启动启动on-hold进程.

## App

对于普通的app进程,跟system\_server进程的启动过来有些类似.不同的是app进程是向发消息给system\_server进程, 由system\_server向zygote发出创建进程的请求.

[理解Android进程创建流程](http://gityuan.com/2016/03/26/app-process-create/), 可知进程创建后 接下来会进入ActivityThread.main()过程。

### ActivityThread.main

|  |
| --- |
| public static void main(String[] args) {  ...  Environment.initForCurrentUser();  ...  Process.setArgV0("<pre-initialized>");  //创建主线程looper  Looper.prepareMainLooper();  ActivityThread thread = new ActivityThread();  thread.attach(false); //attach到系统进程  if (sMainThreadHandler == null) {  sMainThreadHandler = thread.getHandler();  }    //主线程进入循环状态  Looper.loop();  throw new RuntimeException("Main thread loop unexpectedly exited");  } |

app进程的主线程调用栈的栈底如下:

|  |
| --- |
| ...  at android.app.ActivityThread.main(ActivityThread.java:5442)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

跟前面介绍的system\_server进程调用栈对比:

|  |
| --- |
| at com.android.server.SystemServer.main(SystemServer.java:175)  at java.lang.reflect.Method.invoke!(Native method)  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628) |

## startService

/\*\*

\* Starts a service by class name.

\*

\* @return The service instance.

\*/

@SuppressWarnings("unchecked")

public SystemService startService(String className) {

final Class<SystemService> serviceClass;

try {

serviceClass = (Class<SystemService>)Class.forName(className);

} catch (ClassNotFoundException ex) {

Slog.i(TAG, "Starting " + className);

throw new RuntimeException("Failed to create service " + className

+ ": service class not found, usually indicates that the caller should "

+ "have called PackageManager.hasSystemFeature() to check whether the "

+ "feature is available on this device before trying to start the "

+ "services that implement it", ex);

}

return startService(serviceClass);

}

/\*\*

\* Creates and starts a system service. The class must be a subclass of

\* {@link com.android.server.SystemService}.

\*

\* @param serviceClass A Java class that implements the SystemService interface.

\* @return The service instance, never null.

\* @throws RuntimeException if the service fails to start.

\*/

@SuppressWarnings("unchecked")

public <T extends SystemService> T startService(Class<T> serviceClass) {

try {

final String name = serviceClass.getName();

Slog.i(TAG, "Starting " + name);

Trace.traceBegin(Trace.TRACE\_TAG\_SYSTEM\_SERVER, "StartService " + name);

// Create the service.

if (!SystemService.class.isAssignableFrom(serviceClass)) {

throw new RuntimeException("Failed to create " + name

+ ": service must extend " + SystemService.class.getName());

}

final T service;

try {

Constructor<T> constructor = serviceClass.getConstructor(Context.class);

service = constructor.newInstance(mContext);

} catch (InstantiationException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service could not be instantiated", ex);

} catch (IllegalAccessException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service must have a public constructor with a Context argument", ex);

} catch (NoSuchMethodException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service must have a public constructor with a Context argument", ex);

} catch (InvocationTargetException ex) {

throw new RuntimeException("Failed to create service " + name

+ ": service constructor threw an exception", ex);

}

// Register it.

mServices.add(service);

// Start it.

try {

service.onStart();

} catch (RuntimeException ex) {

throw new RuntimeException("Failed to start service " + name

+ ": onStart threw an exception", ex);

}

return service;

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_SYSTEM\_SERVER);

}

}

# AMS onCreate

在android系统中，应用程序是由Activity组成的，因此，应用程序的启动过程实际上就是应用程序中的默认Activity的启动过程，本文将详细分析应用程序框架层的源代码，了解Android应用程序的启动过程。在手机屏幕中点击应用程序图标开始——到应用MainActivity展示出来结束。

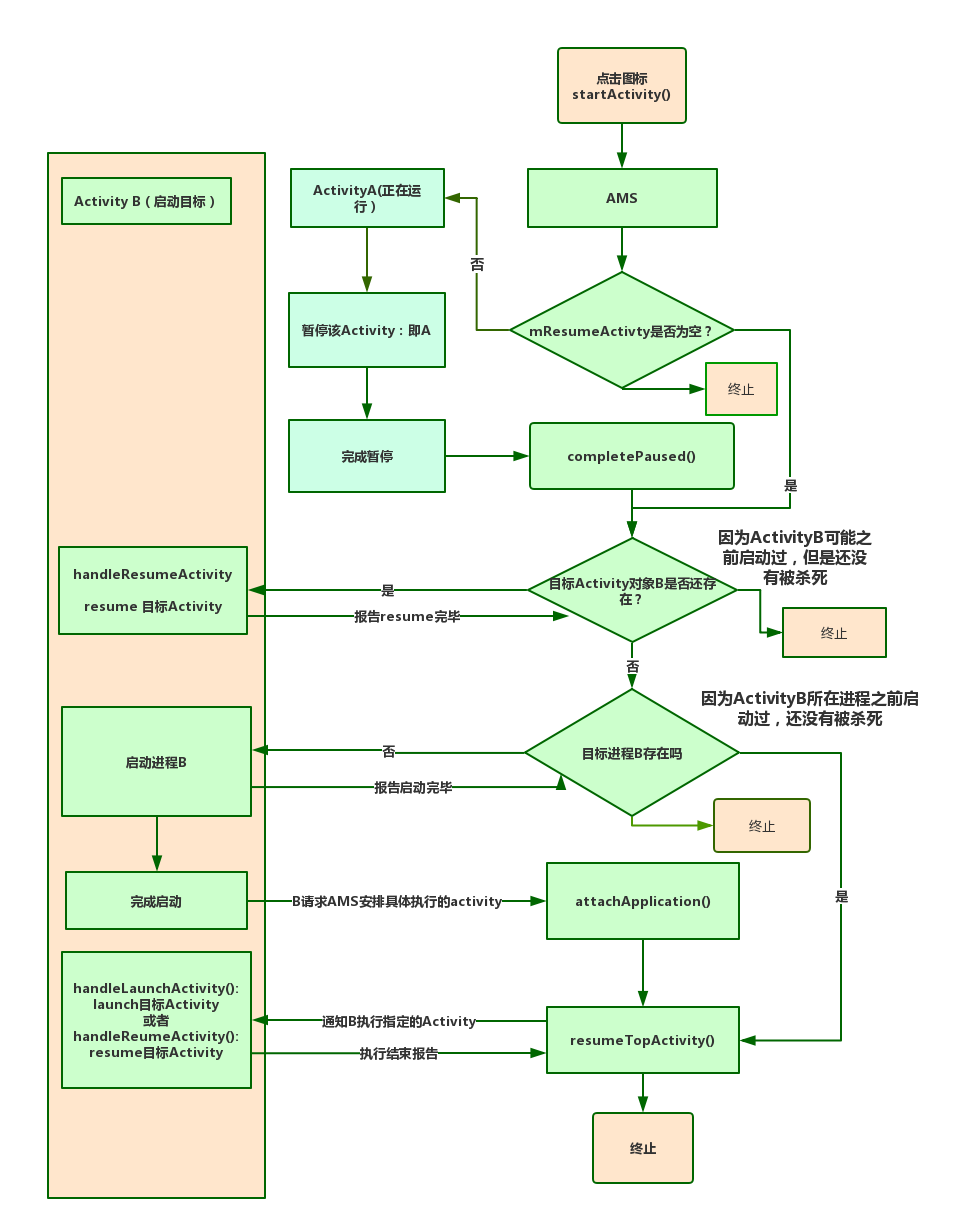
下面详细分析每一步是如何实现的。

## activity的启动流程

activity的启动流程大概总结为

启动Activity B -> 当前有正在显示的activity吗 -> 有就先pause() -> B的进程存在吗 -> 不存在则创建 -> B进程启动指定的Activity

首先来看一下这个过程的具体流程图，其实就是上面流程简介的具体化



## Activity的生命周期

ActivityState定义

**enum** ActivityState {  
 INITIALIZING,  
 RESUMED,  
 PAUSING,  
 PAUSED,  
 STOPPING,  
 STOPPED,  
 FINISHING,  
 DESTROYING,  
 DESTROYED  
}

启动：onCreate -> onStart -> onResume  
退出：onPause -> onStop -> onDestroy  
后台：onPause -> onStop  
返回：onRestart -> onStart -> onResume  
异常情况：  
onSaveInstanceState -> onStop -> onDestroy 和onPause无时序关系  
onCreate -> onStart -> onRestoreInstanceState 和onResume无时序关系

### onStart & onStop 与 onResume & onDestroy区别：

onStart & onStop：表示activity是否可见  
onResume & onPause：表示activity是否位于前台

### 耗时操作

A启动B,A必须先onPause，B才可onResume。所以onPause不要执行重量级操作(耗时操作)，应把重量级操作放到onStop中。

### 1 Activity的生命周期全面分析

正常周期（用户参与）和异常周期（系统回收、Configuration改变）

**正常周期**



总结一下各种情况下Activity生命周期的调用情况。

**1，正常打开单个Activity，然后退出应用：**

这种情况是最普通的状况，Activity的生命周期会按照上图从上到下的方式走。即：onCreate --> onStart --> onResume --> 运行--> 按返回键结束程序--> onPause-->onStop-->onDestory

**2，打开一个Activity A，然后再打开另一个Activity B**

对于A：

onCreate --> onStart --> onResume --> A运行 --> A发出打开B的Intent --> onPause-->B可见-->onStop

此时，会打开B，B同样会经历一个完整的Activity生命周期。

等B结束，A再度可见的时候，A会经历：onRestart-->onStart-->onResume

注意，B这个Activity是在A的onPause执行后才变成可见状态的，所以为了不影响B的显示，最好不要在onPause里执行一些耗时操作，可以考虑将这些操作放到onStop里，这时B已经可见了。

**3，打开一个Activity A，然后再打开另一个Activity B（透明主题）或者是对话框B**

对于A：

onCreate --> onStart --> onResume --> A运行 --> A发出打开B的Intent --> onPause-->B可见--~~>onStop~~

此时，会打开B，B同样会经历一个完整的Activity生命周期。

等B结束，A再度可见的时候，A会经历：~~onRestart-->onStart-->~~onResume

**3，优先级低的Activity被系统回收**

Activity优先级从高到低大致有三种：

1）刚执行onResume，前台Activity。可见的前台Activity优先级最高

2）刚执行onPause，可见，但非前台的Activity。例如弹出了一个对话框，Activity可见，但是不可与用户交互。此类别优先级低一些。

3）刚执行onStop，后台Activity。这个就是不可见的Activity。

系统资源不足时，会优先回收优先级低的Activity。这个时候被结束的Activity，并不会像正常的Activity一样，在销毁的时候会走onPause，onStop，onDestory这样的生命周期，而是会多一个onSaveInstanceState的方法来保存一些数据。这个方法会在onStop前调用，但是不确定是在onPause之前还是之后。Activity重建的时候，这个方法中所保存的数据会以Bundle的形式作为参数传递给onCreate，所以我们可以用参数来判断这个Activity是新建的还是重建的，如果是重建的，则可以从Bundle中取出数据来恢复界面。

**QA：**

**Q1**: onStart和onResume, onPause和onStop从描述上来看差不多，对我们来说有什么实质的不同呢?

**A1：**从整个生命周期来说，onCreate和onDestroy是配对的，分别标识着Activity的创建和销毁，并且只可能有一次调用。从Activity是否可见来说，onStart和onStop是配对的（**可见性状态**），随着用户的操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次;从Activity是否在前台来说，onResume和onPause是配对的（**前后台状态**），随着用户操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次。

从实际使用过程来说，onStart和anResume. onPause和onStop看起来的确差不多，甚至我们可以只保留其中一对，比如只保留onStart和onStop 既然如此，那Android系统还要提供看起来重复的接口呢?这两个配对的回调分别表示不同的意义，onStart和onStop是从Activity是否可见这个角度来回调的，而onResume和onPause。是从Activity是否位于前台这个角度来回调的，除了这种区别，在实际使用中没有其他明显区别。

**Q2:**假设当前Activity为A，如果这时用户打开一个新Activity B，那么B的onResume和A的onPause哪个先执行呢?

**A2：**onPause先执行-》onPause和onStop都不能执行耗时的操作，尤其是onPause，应当尽量在onStop中做操作，从而使得新Activity尽快显示出来并切换到前台。

**A3：**了解这些生命周期

**Q3：**自定义与视图有关的工具的时候，也应该有这些生命周期，比如动画工具

**异常周期**

**系统配置**

在默认情况下，如果我们的Activity不做特殊处理，那么当系统配置发生改变后，Activity就会被销毁并重新创建，其生命周期如图1-3所示。系统只在Activity异常终止的时候才会调用onSaveInstanceState和onRestoreInstanceState来存储和恢复数据，其他情况不会触发这个过程。



当系统配置发生改变后，系统会调用onSaveInstanceState来保存当前Activity的状态。这个方法的调用时机是在onStop之前，它和onPause没有既定的时序关系，可前可后。这个方法只会出现在Activity被异常终止的情况下，正常情况下系统不会回调。当Activity被重新创建后，系统会调用onRestorelnstanceSiate，从时序来说，onRestoreInstanceState的调用时机在onStart之后。

Activity被销毁并重新创建后，我们再去获取之前存储的字符串。接收的位置可以选择onRestorelnstanceState或者onCreate，二者的区别onRestorelnstanceState参数非空，但是onCreate可空，onCreate如果是正常启动的话，其参数Bundle saVedlnstanceState为null,

所以必须要额外判断。这两个方法我们选择任意一个都可以进行数据恢复，但官方建议是采用onRestoreInstanceState去恢复数据

系统配置监听，取代onSaveInstanceState和onRestorelnstanceState

android:configChanges="orientation|screenSize"

@Override

public void onConfigurationChanged(Configuration newConfig) {

super.onConfigurationChanged(newConfig);

Log.d(TAG, "onConfigurationChanged, newOrientation:" + newConfig.orientation);

}

情况2:**资源内存**不足导致低优先级的Activity被杀死

这种情况我们不好模拟，但是其数据存储和恢复过程和情况1完全一致。这里我们描述一下Activity的优先级情况口Activity按照优先级从高到低，可以分为如下3种:

(1)前台Activity—正在和用户交互的Activity，优先级最高。

(2)可见但非前台Activity—比如Activity中弹出了一个对话框，导致Activity可见，但是位于后台无法和用户直接交互。

（3)后台Activity—已经被暂停的Activity，比如执行了onStop，优先级最低。

**小结**：如果一个进程中没有四大组件在执行，那么这个进程将很快被系统杀死，因此，一此后台工作不适合脱离四大组件而独自运行在后台中，这样进程很容易被杀死。比较好的方法是将后台工作放入Service中从而保证进程有一定的优先级，这样就不会轻易地被系统杀死。

## Luancher进程

### Step 1. Launcher.startActivitySafely

1. **void** startActivitySafely(Intent intent, Object tag) {
2. intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);
3. try {
4. startActivity(intent);
5. } catch (ActivityNotFoundException e) {
6. ......
7. } catch (SecurityException e) {
8. ......
9. }
10. }

MainActivity，这里是AndroidManifest.xml文件中配置的：

1. **<activity** android:name=".MainActivity"
2. android:label="@string/app\_name"**>**
3. **<intent-filter>**
4. **<action** android:name="android.intent.action.MAIN" **/>**
5. **<category** android:name="android.intent.category.LAUNCHER" **/>**
6. **</intent-filter>**
7. **</activity>**

因此，这里的intent包含的信息为：action = "android.intent.action.Main"，category="android.intent.category.LAUNCHER", cmp="shy.luo.activity/.MainActivity"，表示它要启动的Activity为shy.luo.activity.MainActivity。Intent.FLAG\_ACTIVITY\_NEW\_TASK表示要在一个新的Task中启动这个Activity，注意，Task是Android系统中的概念，它不同于进程Process的概念。简单地说，一个Task是一系列Activity的集合，这个集合是以堆栈的形式来组织的，遵循后进先出的原则。一个task可以包含多个process。





在Android系统中，应用程序是由Launcher启动起来的，其实，Launcher本身也是一个应用程序，其它的应用程序安装后，就会Launcher的界面上出现一个相应的图标，点击这个图标时，Launcher就会对应的应用程序启动起来。

### Step 2. Activity.startActivity

Launcher的父类Activity实现了startActivity函数，因此，这里就调用了Activity.startActivity函数。 这个函数实现很简单，它调用startActivityForResult来进一步处理，第二个参数传入-1表示不需要这个Actvity结束后的返回结果。

1. @Override
2. **public** **void** startActivity(Intent intent) {
3. startActivityForResult(intent, -1);
4. }

### Step 3. Activity.startActivityForResult

1. **public** **void** startActivityForResult(Intent **intent**, **int** requestCode) {
2. **if** (mParent == **null**) {
3. Instrumentation.ActivityResult ar =
4. mInstrumentation.execStartActivity(
5. **this**, mMainThread.getApplicationThread(), mToken, **this**,
6. **intent**, requestCode);
7. ......
8. } **else** {
9. ......
10. }

这里的mInstrumentation是Activity类的成员变量，它的类型是Intrumentation，定义在frameworks/base/core/java/android/app/Instrumentation.java文件中，它用来监控应用程序和系统的交互。

这里的mMainThread也是Activity类的成员变量，它的类型是ActivityThread，它代表的是应用程序的主线程，这里通过mMainThread.getApplicationThread获得它里面的ApplicationThread成员变量，它是一个Binder对象，后面我们会看到，AtMS会使用它来和ActivityThread来进行进程间通信。这里我们需注意的是，**这里的mMainThread代表的是Launcher应用程序运行的进程。**

这里的mToken也是Activity类的成员变量，它是一个Binder对象的远程接口。

### Step 11. ActivityStack.startPausingLocked

1. **public** **class** ActivityStack {
3. ......
5. **private** **final** **void** startPausingLocked(**boolean** userLeaving, **boolean** uiSleeping) {
6. **if** (mPausingActivity != **null**) {
7. ......
8. }
9. ActivityRecord prev = mResumedActivity;
10. **if** (prev == **null**) {
11. ......
12. }
13. ......
14. mResumedActivity = **null**;
15. mPausingActivity = prev;
16. mLastPausedActivity = prev;
17. prev.state = ActivityState.PAUSING;
18. ......
20. **if** (prev.app != **null** && prev.app.thread != **null**) {
21. ......
22. **try** {
23. ......
24. prev.app.thread.schedulePauseActivity(prev, prev.finishing, userLeaving,
25. prev.configChangeFlags);
26. ......
27. } **catch** (Exception e) {
28. ......
29. }
30. } **else** {
31. ......
32. }
34. ......
36. }
38. ......
40. }

函数首先把mResumedActivity保存在本地变量prev中。在上一步Step 10中，说到mResumedActivity就是Launcher，因此，这里把Launcher进程中的ApplicationThread对象取出来，通过它来通知Launcher这个Activity它要进入Paused状态了。当然，这里的prev.app.thread是一个ApplicationThread对象的远程接口，通过调用这个远程接口的schedulePauseActivity来通知Launcher进入Paused状态。

参数prev.finishing表示prev所代表的Activity是否正在等待结束的Activity列表中，由于Laucher这个Activity还没结束，所以这里为false；参数prev.configChangeFlags表示哪些config发生了变化，这里我们不关心它的值。

### Step 4. Instrumentation.execStartActivity

在frameworks/base/core/java/android/app/Instrumentation.java文件中：

1. **public** **class** Instrumentation {
2. **public** ActivityResult execStartActivity(
3. Context who, IBinder contextThread, IBinder token, Activity target,
4. Intent intent, **int** requestCode) {
5. IApplicationThread whoThread = (IApplicationThread) contextThread;
6. **if** (mActivityMonitors != **null**) {
7. ......
8. }
9. **try** {
10. **int** result = ActivityManagerNative.getDefault()
11. .**startActivity**(whoThread, intent,
12. intent.**resolveTypeIfNeeded**(who.getContentResolver()),
13. **null**, 0, token, target != **null** ? target.mEmbeddedID : **null**,
14. requestCode, **false**, **false**);
15. ......
16. } **catch** (RemoteException e) {
17. }
18. **return** **null**;
19. }
21. }

这里的ActivityManagerNative.getDefault返回**AtMS**的远程接口，即**ActivityManagerProxy**接口，具体可以参考Android系统在新进程中启动自定义服务过程（startService）的原理分析一文。

这里的intent.resolveTypeIfNeeded返回这个intent的MIME类型，在这个例子中，没有AndroidManifest.xml设置MainActivity的MIME类型，因此，这里返回null。

这里的target不为null，但是target.mEmbddedID为null，我们不用关注。

### Step 5. ActivityManagerProxy.startActivity

定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **int** startActivity(IApplicationThread caller, Intent intent,
4. String resolvedType, Uri[] grantedUriPermissions, **int** grantedMode,
5. IBinder resultTo, String resultWho,
6. **int** requestCode, **boolean** onlyIfNeeded,
7. **boolean** debug) **throws** RemoteException {
8. Parcel data = Parcel.obtain();
9. Parcel reply = Parcel.obtain();
10. data.writeInterfaceToken(IActivityManager.descriptor);
11. data.writeStrongBinder(caller != **null** ? caller.asBinder() : **null**);
12. intent.writeToParcel(data, 0);
13. data.writeString(resolvedType);  **//null**
14. data.writeTypedArray(grantedUriPermissions, 0);   **//null**
15. data.writeInt(grantedMode);
16. data.writeStrongBinder(resultTo);
17. data.writeString(resultWho);
18. data.writeInt(requestCode);  //-1
19. data.writeInt(onlyIfNeeded ? 1 : 0);  //false
20. data.writeInt(debug ? 1 : 0);  //false
21. **mRemote**.transact(START\_ACTIVITY\_TRANSACTION, data, reply, 0);
22. reply.readException();
23. **int** result = reply.readInt();
24. reply.recycle();
25. data.recycle();
26. **return** result;
27. }
28. }

这里的参数比较多，我们先整理一下。从上面的调用可以知道，这里的参数resolvedType、grantedUriPermissions和resultWho均为null；参数caller为ApplicationThread类型的Binder实体；参数resultTo为一个Binder实体的远程接口，我们先不关注它；参数grantedMode为0，我们也先不关注它；参数requestCode为-1；参数onlyIfNeeded和debug均空false。**mRemote开始向AMS进程间通信，发送**START\_ACTIVITY\_TRANSACTION消息。

## AMS进程

上述都在launer中处理的，接下是AMS响应START\_ACTIVITY\_TRANSACTIO的进程间通信

### Step 6. AtMS.startActivity

上一步Step 5通过Binder驱动程序就进入到AtMS的startActivity函数来了

@Override  
**public final int** startActivityAsUser(IApplicationThread caller, String callingPackage,  
 Intent intent, String resolvedType, IBinder resultTo, String resultWho, **int** requestCode,  
 **int** startFlags, ProfilerInfo profilerInfo, Bundle bOptions, **int** userId) {  
 enforceNotIsolatedCaller(**"startActivity"**);  
 userId = **mUserController**.handleIncomingUser(Binder.*getCallingPid*(), Binder.*getCallingUid*(),  
 userId, **false**, ***ALLOW\_FULL\_ONLY***, **"startActivity"**, **null**);  
 *//* ***TODO: Switch to user app stacks here.* return mActivityStarter**.startActivityMayWait(caller, -1, callingPackage, intent,  
 resolvedType, **null**, **null**, resultTo, resultWho, requestCode, startFlags,  
 profilerInfo, **null**, **null**, bOptions, **false**, userId, **null**, **null**,  
 **"startActivityAsUser"**);  
}

这里只是简单地将操作转发给成员变量mActivityStarter的startActivityMayWait函数。

### ActivityStarter.startActivityMayWait

2. **final** **int** startActivityMayWait(IApplicationThread caller,
3. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
4. **int** grantedMode, IBinder resultTo,
5. String resultWho, **int** requestCode, **boolean** onlyIfNeeded,
6. **boolean** debug, WaitResult outResult, Configuration config) {
7. **boolean** componentSpecified = intent.getComponent() != **null**;
8. // Don't modify the client's object!
9. intent = **new** Intent(intent);
10. // Collect information about the target of the Intent.
11. ActivityInfo aInfo;
12. **try** {
13. ResolveInfo rInfo =
14. AppGlobals.getPackageManager().resolveIntent(
15. intent, resolvedType,
16. PackageManager.MATCH\_DEFAULT\_ONLY
17. | AtMS.STOCK\_PM\_FLAGS);
18. aInfo = rInfo != **null** ? rInfo.activityInfo : **null**;
19. } **catch** (RemoteException e) {
20. ......
21. }
23. **if** (aInfo != **null**) {
24. // Store the found target back into the intent, because now that
25. // we have it we never want to do this again.  For example, if the
26. // user navigates back to this point in the history, we should
27. // always restart the exact same activity.
28. intent.setComponent(**new** ComponentName(
29. aInfo.applicationInfo.packageName, aInfo.name));
30. ......
31. }
33. **synchronized** (mService) {
34. **int** callingPid;
35. **int** callingUid;
36. **if** (caller == **null**) {
37. ......
38. } **else** {
39. callingPid = callingUid = -1;
40. }
42. mConfigWillChange = config != **null**
43. && mService.mConfiguration.diff(config) != 0;
44. **int** res = **startActivityLocked**(caller, intent, resolvedType,
45. grantedUriPermissions, grantedMode, aInfo,
46. resultTo, resultWho, requestCode, callingPid, callingUid,
47. onlyIfNeeded, componentSpecified);
48. **return** res;
49. }

注意，从Step 6传下来的参数outResult和config均为null，此外，表达式(aInfo.applicationInfo.flags&ApplicationInfo.FLAG\_CANT\_SAVE\_STATE) != 0为false，因此，这里忽略了无关代码。 下面语句对参数intent的内容进行解析，得到MainActivity的相关信息，保存在aInfo变量中：

#### resolveActivity

hgk

*// Collect information about the target of the Intent.*ActivityInfo aInfo = **mSupervisor**.resolveActivity(intent, rInfo, startFlags, profilerInfo);

**final** ActivityRecord[] outRecord = **new** ActivityRecord[1];  
**int** res = startActivityLocked(caller, intent, ephemeralIntent, resolvedType,  
 aInfo, rInfo, voiceSession, voiceInteractor,  
 resultTo, resultWho, requestCode, callingPid,  
 callingUid, callingPackage, realCallingPid, realCallingUid, startFlags,  
 options, ignoreTargetSecurity, componentSpecified, outRecord, container,  
 inTask, reason);

解析之后，得到的aInfo.applicationInfo.packageName的值为"k.demo.asys"，aInfo.name的值为"k.demo.asys.activity.MainActivity"，这是在这个实例的配置文件AndroidManifest.xml里面配置的。此外，函数开始的地方调用intent.getComponent()函数的返回值不为null，因此，这里的componentSpecified变量为true。

接下去就调用startActivityLocked进一步处理了。

### Step 8. ActivityStarter.startActivityLocked

这个函数：

**mLastStartActivityResult** = startActivity(caller, intent, ephemeralIntent, resolvedType,  
 aInfo, rInfo, voiceSession, voiceInteractor, resultTo, resultWho, requestCode,  
 callingPid, callingUid, callingPackage, realCallingPid, realCallingUid, startFlags,  
 options, ignoreTargetSecurity, componentSpecified, **mLastStartActivityRecord**,  
 container, inTask);

#### startActivity

**private int** startActivity(IApplicationThread caller, Intent intent, Intent ephemeralIntent,  
 String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,  
 IBinder resultTo, String resultWho, **int** requestCode, **int** callingPid, **int** callingUid,  
 String callingPackage, **int** realCallingPid, **int** realCallingUid, **int** startFlags,  
 ActivityOptions options, **boolean** ignoreTargetSecurity, **boolean** componentSpecified,  
 ActivityRecord[] outActivity, ActivityStackSupervisor.ActivityContainer container,  
 TaskRecord inTask) {

**final int** launchFlags = intent.getFlags();

ActivityRecord r = **new** ActivityRecord(**mService**, callerApp, callingPid, callingUid,  
 callingPackage, intent, resolvedType, aInfo, **mService**.getGlobalConfiguration(),  
 resultRecord, resultWho, requestCode, componentSpecified, voiceSession != **null**,  
 **mSupervisor**, container, options, sourceRecord);

**return** startActivity(r, sourceRecord, voiceSession, voiceInteractor, startFlags, **true**,  
 options, inTask, outActivity);

从传进来的参数caller得到调用者的进程信息，并保存在callerApp变量中，这里就是Launcher应用程序的进程信息了。前面说过，参数resultTo是Launcher这个Activity里面的一个Binder对象，通过它可以获得Launcher这个Activity的相关信息，保存在sourceRecord变量中。再接下来，创建即将要启动的Activity的相关信息，并保存在r变量中：

1. ActivityRecord r = **new** ActivityRecord(mService, **this**, callerApp, callingUid,
2. intent, resolvedType, aInfo, mService.mConfiguration,
3. resultRecord, resultWho, requestCode, componentSpecified);

#### startActivity

**private int** startActivity(**final** ActivityRecord r, ActivityRecord sourceRecord,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,  
 **int** startFlags, **boolean** doResume, ActivityOptions options, TaskRecord inTask,  
 ActivityRecord[] outActivity) {

{  
 **int** result = START\_CANCELED;  
 **try** {  
 **mService**.**mWindowManager**.deferSurfaceLayout();  
 result = startActivityUnchecked(r, sourceRecord, voiceSession, voiceInteractor,  
 startFlags, doResume, options, inTask, outActivity);  
 } **finally** {  
 *// If we are not able to proceed, disassociate the activity from the task. Leaving an  
 // activity in an incomplete state can lead to issues, such as performing operations  
 // without a window container.* **if** (!ActivityManager.isStartResultSuccessful(result)  
 && **mStartActivity**.getTask() != **null**) {  
 **mStartActivity**.getTask().removeActivity(**mStartActivity**);  
 }  
 **mService**.**mWindowManager**.continueSurfaceLayout();  
 }  
  
 postStartActivityProcessing(r, result, **mSupervisor**.getLastStack().**mStackId**, **mSourceRecord**,  
 **mTargetStack**);  
  
 **return** result;  
}

### startActivityUnchecked

**private int** startActivityUnchecked(**final** ActivityRecord r, ActivityRecord sourceRecord,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,  
 **int** startFlags, **boolean** doResume, ActivityOptions options, TaskRecord inTask,  
 ActivityRecord[] outActivity) {

computeLaunchingTaskFlags();  
computeSourceStack();  
  
**mIntent**.setFlags(**mLaunchFlags**);

ActivityRecord reusedActivity = getReusableIntentActivity();

**if** (!**mAddingToTask** && **mReuseTask** == **null**) {  
 *// We didn't do anything... but it was needed (a.k.a., client don't use that  
 // intent!) And for paranoia, make sure we have correctly resumed the top activity.* resumeTargetStackIfNeeded();  
 **if** (outActivity != **null** && outActivity.**length** > 0) {  
 outActivity[0] = reusedActivity;  
 }  
 **return** START\_TASK\_TO\_FRONT;  
}

#### computeLaunchingTaskFlags();

#### getReusableIntentActivit

**private** ActivityRecord getReusableIntentActivity() {  
 *// We may want to try to place the new activity in to an existing task. We always  
 // do this if the target activity is singleTask or singleInstance; we will also do  
 // this if NEW\_TASK has been requested, and there is not an additional qualifier telling  
 // us to still place it in a new task: multi task, always doc mode, or being asked to  
 // launch this as a new task behind the current one.* **boolean** putIntoExistingTask = ((**mLaunchFlags** & ***FLAG\_ACTIVITY\_NEW\_TASK***) != 0 &&  
 (**mLaunchFlags** & ***FLAG\_ACTIVITY\_MULTIPLE\_TASK***) == 0)  
 || **mLaunchSingleInstance** || **mLaunchSingleTask**;  
 *// If bring to front is requested, and no result is requested and we have not been given  
 // an explicit task to launch in to, and we can find a task that was started with this  
 // same component, then instead of launching bring that one to the front.* putIntoExistingTask &= **mInTask** == **null** && **mStartActivity**.**resultTo** == **null**;  
 ActivityRecord intentActivity = **null**;  
 **if** (**mOptions** != **null** && **mOptions**.getLaunchTaskId() != -1) {  
 **final** TaskRecord task = **mSupervisor**.anyTaskForIdLocked(**mOptions**.getLaunchTaskId());  
 intentActivity = task != **null** ? task.getTopActivity() : **null**;  
 } **else if** (putIntoExistingTask) {  
 **if** (**mLaunchSingleInstance**) {  
 *// There can be one and only one instance of single instance activity in the  
 // history, and it is always in its own unique task, so we do a special search.* intentActivity = **mSupervisor**.findActivityLocked(**mIntent**, **mStartActivity**.**info**, **false**);  
 } **else if** ((**mLaunchFlags** & ***FLAG\_ACTIVITY\_LAUNCH\_ADJACENT***) != 0) {  
 *// For the launch adjacent case we only want to put the activity in an existing  
 // task if the activity already exists in the history.* intentActivity = **mSupervisor**.findActivityLocked(**mIntent**, **mStartActivity**.**info**,  
 !**mLaunchSingleTask**);  
 } **else** {  
 *// Otherwise find the best task to put the activity in.* intentActivity = **mSupervisor**.findTaskLocked(**mStartActivity**, **mSourceDisplayId**);  
 }  
 }  
 **return** intentActivity;  
}

接着调用startActivityUncheckedLocked函数进行下一步操作。

### Step 9. ActivityStarter.startActivityUncheckedLocked

这：

1. **public** **class**
2. **final** **int** startActivityUncheckedLocked(ActivityRecord r,
3. ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
4. **int** grantedMode, **boolean** onlyIfNeeded, **boolean** doResume) {
5. **final** Intent intent = r.intent;
6. **final** **int** callingUid = r.launchedFromUid;
7. **int** launchFlags = intent.getFlags();
8. // We'll invoke onUserLeaving before onPause only if the launching
9. // activity did not explicitly state that this is an automated launch.
10. mUserLeaving = (launchFlags&Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION) == 0;
11. ActivityRecord notTop = (launchFlags&Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP)
12. != 0 ? r : **null**;
13. // If the onlyIfNeeded flag is set, then we can do this if the activity
14. // being launched is the same as the one making the call...  or, as
15. // a special case, if we do not know the caller then we count the
16. // current top activity as the caller.
17. **if** (onlyIfNeeded) {
18. ......
19. }
21. **if** (sourceRecord == **null**) {
22. ......
23. } **else** **if** (sourceRecord.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
24. ......
25. } **else** **if** (r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
26. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK) {
27. ......
28. }
30. **if** (r.resultTo != **null** && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
31. ......
32. }
34. **boolean** addingToTask = **false**;
35. **if** (((**launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK)** != 0 &&
36. (launchFlags&Intent.**FLAG\_ACTIVITY\_MULTIPLE\_TASK**) == 0)
37. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
38. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
39. // If bring to front is requested, and no result is requested, and
40. // we can find a task that was started with this same
41. // component, then instead of launching bring that one to the front.
42. **if** (r.resultTo == **null**) {
43. // See if there is a task to bring to the front.  If this is
44. // a SINGLE\_INSTANCE activity, there can be one and only one
45. // instance of it in the history, and it is always in its own
46. // unique task, so we do a special search.
47. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
48. ? findTaskLocked(intent, r.info)
49. : findActivityLocked(intent, r.info);
50. **if** (taskTop != **null**) {
51. ......
52. }
53. }
54. }
55. **if** (r.packageName != **null**) {
56. // If the activity being launched is the same as the one currently
57. // at the top, then we need to check if it should only be launched
58. // once.
59. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
60. **if** (top != **null** && r.resultTo == **null**) {
61. **if** (top.realActivity.equals(r.realActivity)) {
62. ......
63. }
64. }
66. }
68. **boolean** newTask = **false**;
70. // Should this be considered a new task?
71. **if** (r.resultTo == **null** && !addingToTask
72. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
73. // todo: should do better management of integers.
74. mService.mCurTask++;
75. **if** (mService.mCurTask <= 0) {
76. mService.mCurTask = 1;
77. }
78. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
79. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
80. ......
81. newTask = **true**;
82. **if** (mMainStack) {
83. mService.addRecentTaskLocked(r.task);
84. }
85. }
86. **startActivityLocked**(r, newTask, doResume);
87. **return** START\_SUCCESS;
88. }
89. }

函数首先获得intent的标志值，保存在launchFlags变量中。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION没有置位，因此 ，成员变量mUserLeaving的值为true。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP也没有置位，因此，变量notTop的值为null。

由于在这个例子的AndroidManifest.xml文件中，MainActivity没有配置launchMode属值，因此，这里的r.launchMode为默认值0，表示以标准（Standard，或者称为ActivityInfo.LAUNCH\_MULTIPLE）的方式来启动这个Activity。Activity的启动方式有四种，其余三种分别是ActivityInfo.LAUNCH\_SINGLE\_INSTANCE、ActivityInfo.LAUNCH\_SINGLE\_TASK和ActivityInfo.LAUNCH\_SINGLE\_TOP，具体可以参考官方网站<http://developer.android.com/reference/android/content/pm/ActivityInfo.html>。

传进来的参数r.resultTo为null，表示Launcher不需要等这个即将要启动的MainActivity的执行结果。

由于这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NEW\_TASK被置位，而且Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK没有置位，因此，下面的if语句会被执行：

1. **if** (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&
2. (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)
3. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
4. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
5. // If bring to front is requested, and no result is requested, and
6. // we can find a task that was started with this same
7. // component, then instead of launching bring that one to the front.
8. **if** (r.resultTo == **null**) {
9. // See if there is a task to bring to the front.  If this is
10. // a SINGLE\_INSTANCE activity, there can be one and only one
11. // instance of it in the history, and it is always in its own
12. // unique task, so we do a special search.
13. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
14. ? findTaskLocked(intent, r.info)
15. : findActivityLocked(intent, r.info);
16. **if** (taskTop != **null**) {
17. ......
18. }
19. }
20. }

这段代码的逻辑是查看一下，当前有没有Task可以用来执行这个Activity。由于r.launchMode的值不为ActivityInfo.LAUNCH\_SINGLE\_INSTANCE，因此，它通过findTaskLocked函数来查找存不存这样的Task，这里返回的结果是null，即taskTop为null，因此，需要创建一个新的Task来启动这个Activity。

接着往下看：

1. **if** (r.packageName != **null**) {
2. // If the activity being launched is the same as the one currently
3. // at the top, then we need to check if it should only be launched
4. // once.
5. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
6. **if** (top != **null** && r.resultTo == **null**) {
7. **if** (top.realActivity.equals(r.realActivity)) {
8. ......
9. }
10. }
12. }

这段代码的逻辑是看一下，当前在堆栈顶端的Activity是否就是即将要启动的Activity，有些情况下，如果即将要启动的Activity就在堆栈的顶端，那么，就不会重新启动这个Activity的别一个实例了，具体可以参考官方网站http://developer.android.com/reference/android/content/pm/ActivityInfo.html。现在处理堆栈顶端的Activity是Launcher，与我们即将要启动的MainActivity不是同一个Activity，因此，这里不用进一步处理上述介绍的情况。

执行到这里，我们知道，要在一个新的Task里面来启动这个Activity了，于是新创建一个Task：

1. **if** (r.resultTo == **null** && !addingToTask
2. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
3. // todo: should do better management of integers.
4. mService.mCurTask++;
5. **if** (mService.mCurTask <= 0) {
6. mService.mCurTask = 1;
7. }
8. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
9. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
10. ......
11. newTask = **true**;
12. **if** (mMainStack) {
13. mService.addRecentTaskLocked(r.task);
14. }
16. }

新建的Task保存在r.task域中，同时，添加到mService中去，这里的mService就是AtMS了。

最后就进入startActivityLocked(r, newTask, doResume)进一步处理了。这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **private** **final** **void** **startActivityLocked**(ActivityRecord r, **boolean** newTask,
3. **boolean** doResume) {
4. **final** **int** NH = mHistory.size();
5. **int** addPos = -1;
6. **if** (!newTask) {
7. ......
8. }
9. // Place a new activity at top of stack, so it is next to interact
10. // with the user.
11. **if** (addPos < 0) {
12. addPos = NH;
13. }
14. // If we are not placing the new activity frontmost, we do not want
15. // to deliver the onUserLeaving callback to the actual frontmost
16. // activity
17. **if** (addPos < NH) {
18. ......
19. }
21. // Slot the activity into the history stack and proceed
22. mHistory.add(addPos, r);
23. r.inHistory = **true**;
24. r.frontOfTask = newTask;
25. r.task.numActivities++;
26. **if** (NH > 0) {
27. // We want to show the starting preview window if we are
28. // switching to a new task, or the next activity's process is
29. // not currently running.
30. ......
31. } **else** {
32. // If this is the first activity, don't do any fancy animations,
33. // because there is nothing for it to animate on top of.
34. ......
35. }
36. **if** (doResume) {
37. resumeTopActivityLocked(**null**);
38. }
39. }
40. }

这里的NH表示当前系统中历史任务的个数，这里肯定是大于0，因为Launcher已经跑起来了。当NH>0时，并且现在要切换新任务时，要做一些任务切的界面操作，这段代码我们就不看了，这里不会影响到下面启Activity的过程，有兴趣的读取可以自己研究一下。

这里传进来的参数doResume为true，于是调用resumeTopActivityLocked进一步操作。

### Step 10. ActivityStack.resumeTopActivityLocked

1. **public** **class** ActivityStack {
2. /\*\*
3. \* Ensure that the top activity in the stack is resumed.
4. \*
5. \* @param prev The previously resumed activity, for when in the process
6. \* of pausing; can be null to call from elsewhere.
7. \*
8. \* @return Returns true if something is being resumed, or false if
9. \* nothing happened.
10. \*/
11. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
12. // Find the first activity that is not finishing.
13. ActivityRecord next = topRunningActivityLocked(**null**);
15. // Remember how we'll process this pause/resume situation, and ensure
16. // that the state is reset however we wind up proceeding.
17. **final** **boolean** userLeaving = mUserLeaving;
18. mUserLeaving = **false**;
20. **if** (next == **null**) {
21. ......
22. }
24. next.delayedResume = **false**;
26. // If the top activity is the resumed one, nothing to do.
27. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
28. ......
29. }
31. // If we are sleeping, and there is no resumed activity, and the top
32. // activity is paused, well that is the state we want.
33. **if** ((mService.mSleeping || mService.mShuttingDown)
34. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
35. ......
36. }
38. ......
40. // If we are currently pausing an activity, then don't do anything
41. // until that is done.
42. **if** (mPausingActivity != **null**) {
43. ......
44. }
46. ......
48. // We need to start pausing the current activity so the top one
49. // can be resumed...
50. **if** (mResumedActivity != **null**) {
51. ......
52. startPausingLocked(userLeaving, **false**);
53. **return** **true**;
54. }
56. ......
57. }
59. ......
61. }

函数先通过调用topRunningActivityLocked函数获得堆栈顶端的Activity，这里就是**MainActivity**了，这是在上面的Step 9设置好的，保存在next变量中。

接下来把mUserLeaving的保存在本地变量userLeaving中，然后重新设置为false，在上面的Step 9中，mUserLeaving的值为true，因此，这里的userLeaving为true。

这里的mResumedActivity为Launcher，因为Launcher是当前正被执行的Activity。

当我们处理休眠状态时，mLastPausedActivity保存堆栈顶端的Activity，因为当前不是休眠状态，所以mLastPausedActivity为null。

有了这些信息之后，下面的语句就容易理解了：

1. // If the top activity is the resumed one, nothing to do.
2. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
3. ......
4. }
6. // If we are sleeping, and there is no resumed activity, and the top
7. // activity is paused, well that is the state we want.
8. **if** ((mService.mSleeping || mService.mShuttingDown)
9. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
10. ......
11. }

就可以了；否则再看一下系统当前是否休眠状态，如果是的话，再看看要启动的Activity是否就是当前处于堆栈顶端的Activity，如果是的话，也是什么都不用做。

上面两个条件都不满足，因此，在继续往下执行之前，首先要把当处于Resumed状态的Activity推入Paused状态，然后才可以启动新的Activity。但是在将当前这个Resumed状态的Activity推入Paused状态之前，首先要看一下当前是否有Activity正在进入Pausing状态，如果有的话，当前这个Resumed状态的Activity就要稍后才能进入Paused状态了，这样就保证了所有需要进入Paused状态的Activity串行处理。

这里没有处于Pausing状态的Activity，即mPausingActivity为null，而且mResumedActivity也不为null，于是就调用startPausingLocked函数把Launcher推入Paused状态去了。

### Step 12. ApplicationThreadProxy.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
3. ......
5. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
6. **boolean** userLeaving, **int** configChanges) **throws** RemoteException {
7. Parcel data = Parcel.obtain();
8. data.writeInterfaceToken(IApplicationThread.descriptor);
9. data.writeStrongBinder(token);
10. data.writeInt(finished ? 1 : 0);
11. data.writeInt(userLeaving ? 1 :0);
12. data.writeInt(configChanges);
13. mRemote.transact(SCHEDULE\_PAUSE\_ACTIVITY\_TRANSACTION, data, **null**,
14. IBinder.FLAG\_ONEWAY);
15. data.recycle();
16. }
18. ......
20. }

这个函数通过Binder进程间通信机制进入到ApplicationThread.schedulePauseActivity函数中。

## L进程

### Step 13. ApplicationThread.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中，它是ActivityThread的内部类：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
3. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
4. **boolean** userLeaving, **int** configChanges) {
5. queueOrSendMessage(
6. finished ? H.PAUSE\_ACTIVITY\_FINISHING : H.PAUSE\_ACTIVITY,
7. token,
8. (userLeaving ? 1 : 0),
9. configChanges);
10. }
11. }
12. }

这里调用的函数queueOrSendMessage是ActivityThread类的成员函数。

上面说到，这里的finished值为false，因此，queueOrSendMessage的第一个参数值为H.PAUSE\_ACTIVITY，表示要暂停token所代表的Activity，即Launcher。

### Step 14. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1) {
3. queueOrSendMessage(what, obj, arg1, 0);
4. }
5. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
6. **synchronized** (**this**) {
7. ......
8. Message msg = Message.obtain();
9. msg.what = what;
10. msg.obj = obj;
11. msg.arg1 = arg1;
12. msg.arg2 = arg2;
13. mH.sendMessage(msg);
14. }
15. }

这里首先将相关信息组装成一个msg，然后通过mH成员变量发送出去，mH的类型是H，继承于Handler类，是ActivityThread的内部类，因此，这个消息最后由H.handleMessage来处理。

### Step 15. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** H **extends** Handler {
3. **public** **void** handleMessage(Message msg) {
4. **switch** (msg.what) {
5. **case** PAUSE\_ACTIVITY:
6. handlePauseActivity((IBinder)msg.obj, **false**, msg.arg1 != 0, msg.arg2);
7. maybeSnapshot();
8. **break**;
9. }

这里调用ActivityThread.handlePauseActivity进一步操作，msg.obj是一个ActivityRecord对象的引用，它代表的是Launcher这个Activity。

### Step 16. ActivityThread.handlePauseActivity

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** handlePauseActivity(IBinder token, **boolean** finished,
3. **boolean** userLeaving, **int** configChanges) {
4. ActivityClientRecord r = mActivities.get(token);
5. **if** (r != **null**) {
6. //Slog.v(TAG, "userLeaving=" + userLeaving + " handling pause of " + r);
7. **if** (userLeaving) {
8. performUserLeavingActivity(r);
9. }
10. r.activity.mConfigChangeFlags |= configChanges;
11. Bundle state = **performPauseActivity**(token, finished, **true**);
12. // Make sure any pending writes are now committed.
13. QueuedWork.waitToFinish();
14. // Tell the activity manager we have paused.
15. **try** {
16. ActivityManagerNative.getDefault().**activityPaused**(token, state);
17. } **catch** (RemoteException ex) {
18. }
19. }
20. }
21. }

函数首先将Binder引用token转换成ActivityRecord的远程接口ActivityClientRecord，然后做了三个事情：1. 如果userLeaving为true，则通过调用performUserLeavingActivity函数来调用Activity.onUserLeaveHint通知Activity，用户要离开它了；2. 调用**performPauseActivity函数来调用Activity.onPause函数**，我们知道，在Activity的生命周期中，当它要让位于其它的Activity时，系统就会调用它的onPause函数；3. 它通知AtMS，这个Activity已经进入Paused状态了，AtMS现在可以完成未竟的事情，即启动MainActivity了。

### Step 17. ActivityManagerProxy.activityPaused

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. ......
5. **public** **void** activityPaused(IBinder token, Bundle state) **throws** RemoteException
6. {
7. Parcel data = Parcel.obtain();
8. Parcel reply = Parcel.obtain();
9. data.writeInterfaceToken(IActivityManager.descriptor);
10. data.writeStrongBinder(token);
11. data.writeBundle(state);
12. mRemote.transact(ACTIVITY\_PAUSED\_TRANSACTION, data, reply, 0);
13. reply.readException();
14. data.recycle();
15. reply.recycle();
16. }
18. ......
20. }

这里通过Binder进程间通信机制就进入到AtMS.activityPaused函数中去了。

## AMS

### Step 18. AtMS.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/AtMS.java文件中：

@Override  
**public final void** activityPaused(IBinder token) {  
 **final long** origId = Binder.*clearCallingIdentity*();  
 **synchronized**(**this**) {  
 ActivityStack stack = ActivityRecord.*getStackLocked*(token);  
 **if** (stack != **null**) {  
 stack.activityPausedLocked(token, **false**);  
 }  
 }  
 Binder.*restoreCallingIdentity*(origId);  
}

这里，又再次进入到ActivityStack类中，执行activityPaused函数。

这里判断app切换就好了。

stack.**mPausingActivity**.processName

### Step 19. ActivityStack.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **void** activityPaused(IBinder token, Bundle icicle, **boolean** timeout) {
3. ActivityRecord r = **null**;
4. **synchronized** (mService) {
5. **int** index = indexOfTokenLocked(token);
6. **if** (index >= 0) {
7. r = (ActivityRecord)mHistory.get(index);
8. **if** (!timeout) {
9. r.icicle = icicle;
10. r.haveState = **true**;
11. }
12. mHandler.removeMessages(PAUSE\_TIMEOUT\_MSG, r);
13. **if** (mPausingActivity == r) {
14. r.state = ActivityState.PAUSED;
15. completePauseLocked();
16. } **else** {
17. ......
18. }
19. }
20. }
21. }
22. }

这里通过参数token在mHistory列表中得到ActivityRecord，从上面我们知道，这个ActivityRecord代表的是Launcher这个Activity，而我们在Step 11中，把Launcher这个Activity的信息保存在mPausingActivity中，因此，这里mPausingActivity等于r，于是，执行completePauseLocked操作。

### Step 20. ActivityStack.completePauseLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** completePauseLocked() {
3. ActivityRecord prev = mPausingActivity;
4. **if** (prev != **null**) {
5. mPausingActivity = **null**;
6. }
7. **if** (!mService.mSleeping && !mService.mShuttingDown) {
8. resumeTopActivityLocked(prev);
9. } **else** {
10. }
11. }
12. }

函数首先把mPausingActivity变量清空，因为现在不需要它了，然后调用resumeTopActivityLokced进一步操作，它传入的参数即为代表Launcher这个Activity的ActivityRecord。

### Step 21. ActivityStack.resumeTopActivityLokced

1. **public** **class** ActivityStack {
2. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
3. // Find the first activity that is not finishing.
4. ActivityRecord next = topRunningActivityLocked(**null**);
6. // Remember how we'll process this pause/resume situation, and ensure
7. // that the state is reset however we wind up proceeding.
8. **final** **boolean** userLeaving = mUserLeaving;
9. mUserLeaving = **false**;
10. next.delayedResume = **false**;
12. // If the top activity is the resumed one, nothing to do.
13. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
14. ......
15. **return** **false**;
16. }
18. // If we are sleeping, and there is no resumed activity, and the top
19. // activity is paused, well that is the state we want.
20. **if** ((mService.mSleeping || mService.mShuttingDown)
21. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
22. ......
23. **return** **false**;
24. }
26. .......

29. // We need to start pausing the current activity so the top one
30. // can be resumed...
31. **if** (mResumedActivity != **null**) {
32. ......
33. **return** **true**;
34. }
35. **if** (next.app != **null** && next.app.thread != **null**) {
36. } **else** {
37. startSpecificActivityLocked(next, **true**, **true**);
38. }
40. **return** **true**;
41. }
42. }

通过上面的Step 9，我们知道，**当前在堆栈顶端的Activity为我们即将要启动的MainActivity**，这里通过调用topRunningActivityLocked将它取回来，保存在next变量中。之前最后一个Resumed状态的Activity，即Launcher，到了这里已经处于Paused状态了，因此，mResumedActivity为null。最后一个处于Paused状态的Activity为Launcher，因此，这里的mLastPausedActivity就为Launcher。前面我们为MainActivity创建了ActivityRecord后，它的app域一直保持为null。有了这些信息后，上面这段代码就容易理解了，它最终调用startSpecificActivityLocked进行下一步操作。

### Step 22. ActivityStack.startSpecificActivityLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** startSpecificActivityLocked(ActivityRecord r,
3. **boolean** andResume, **boolean** checkConfig) {
4. // Is this activity's application already running?
5. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
6. r.info.applicationInfo.uid);
7. **if** (app != **null** && app.thread != **null**) {
8. **try** {
9. realStartActivityLocked(r, app, andResume, checkConfig);
10. **return**;
11. } **catch** (RemoteException e) {
12. ......
13. }
14. }
16. mService.startProcessLocked(r.processName, r.info.applicationInfo, **true**, 0,
17. "activity", r.intent.getComponent(), **false**);
18. }
19. }

注意，这里由于是第一次启动应用程序的Activity，所以下面语句：

ProcessRecord app = mService.getProcessRecordLocked(r.processName, r.info.applicationInfo.uid); 取回来的app为null。

在Activity应用程序中的AndroidManifest.xml配置文件中，我们没有指定Application标签的process属性，系统就会默认使用package的名称，这里就是"k.demo.system"了。每一个应用程序都有自己的uid，因此，这里uid + process的组合就可以为每一个应用程序创建一个ProcessRecord。当然，我们可以配置两个应用程序具有相同的uid和package，或者在AndroidManifest.xml配置文件的application标签或者activity标签中显式指定相同的process属性值，这样，不同的应用程序也可以在同一个进程中启动。

函数最终执行AtMS.startProcessLocked函数进行下一步操作。

### Step 23. AtMS.startProcessLocked

这个函数定

**final** ProcessRecord startProcessLocked(String processName,

1. ApplicationInfo info, **boolean** knownToBeDead, **int** intentFlags,
2. String hostingType, ComponentName hostingName, **boolean** allowWhileBooting) {
3. ProcessRecord app = getProcessRecordLocked(processName, info.uid);
4. String hostingNameStr = hostingName != **null**
5. ? hostingName.flattenToShortString() : **null**;
6. **if** (app == **null**) {
7. app = **new** ProcessRecordLocked(**null**, info, processName);
8. mProcessNames.put(processName, info.uid, app);
9. } **else** {
10. // If this is a new package in the process, add the package to the list
11. app.addPackage(info.packageName);
12. }
13. startProcessLocked(app, hostingType, hostingNameStr);
14. **return** (app.pid != 0) ? app : **null**;
15. }
16. }

这里再次检查是否已经有以process + uid命名的进程存在，在我们这个情景中，返回值app为null，因此，后面会创建一个ProcessRecord，并存保存在成员变量mProcessNames中，最后，调用另一个startProcessLocked函数进一步操作：

#### startProcessLocked

这里调用Process.start函数创建了一个新的进程，指定新的进程执行android.app.ActivityThread类。最后将表示这个新进程的ProcessRecord保存在mPidSelfLocked列表中，后面会用到。

1. **private** **final** **void** startProcessLocked(ProcessRecord app,
2. String hostingType, String hostingNameStr) {
3. **if** (entryPoint == **null**) entryPoint = **"android.app.ActivityThread"**;  
   Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, **"Start proc: "** +  
    app.processName);  
   checkTime(startTime, **"startProcess: asking zygote to start proc"**);  
   ProcessStartResult startResult;  
   **if** (hostingType.equals(**"webview\_service"**)) {  
    startResult = startWebView(entryPoint,  
    app.processName, uid, uid, gids, debugFlags, mountExternal,  
    app.info.targetSdkVersion, seInfo, requiredAbi, instructionSet,  
    app.info.dataDir, **null**, entryPointArgs);  
   } **else** {  
    startResult = Process.start(entryPoint,  
    app.processName, uid, uid, gids, debugFlags, mountExternal,  
    app.info.targetSdkVersion, seInfo, requiredAbi, instructionSet,  
    app.info.dataDir, invokeWith, entryPointArgs);  
   }

**synchronized** (mPidsSelfLocked) {  
 **this**.mPidsSelfLocked.put(startResult.pid, app);  
 **if** (isActivityProcess) {  
 Message msg = mHandler.obtainMessage(PROC\_START\_TIMEOUT\_MSG);  
 msg.obj = app;  
 mHandler.sendMessageDelayed(msg, startResult.usingWrapper  
 ? PROC\_START\_TIMEOUT\_WITH\_WRAPPER : PROC\_START\_TIMEOUT);  
 }  
}

1. }

#### Process.start

这里主要是调用**Process.start**接口来创建一个新的进程，新的进程会导入android.app.ActivityThread类，并且执行它的main函数，这就是为什么我们前面说每一个应用程序都有一个ActivityThread实例来对应的原因。

Process.ProcessStartResult startResult = Process.start(entryPoint,

app.processName, uid, uid, gids, debugFlags, mountExternal,

app.info.targetSdkVersion, app.info.seinfo, requiredAbi, instructionSet,

app.info.dataDir, entryPointArgs);

#### ZygoteProcess. start

**public final** Process.ProcessStartResult start(**final** String processClass,  
 **final** String niceName,  
 **int** uid, **int** gid, **int**[] gids,  
 **int** debugFlags, **int** mountExternal,  
 **int** targetSdkVersion,  
 String seInfo,  
 String abi,  
 String instructionSet,  
 String appDataDir,  
 String invokeWith,  
 String[] zygoteArgs) {  
 **try** {  
 **return** startViaZygote(processClass, niceName, uid, gid, gids,  
 debugFlags, mountExternal, targetSdkVersion, seInfo,  
 abi, instructionSet, appDataDir, invokeWith, zygoteArgs);  
 } **catch** (ZygoteStartFailedEx ex) {  
 Log.e(LOG\_TAG,  
 **"Starting VM process through Zygote failed"**);  
 **throw new** RuntimeException(  
 **"Starting VM process through Zygote failed"**, ex);  
 }  
}

### Step 24. ActivityThread.main

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** attach(**boolean** system) {
3. mSystemThread = system;
4. **if** (!system) {
5. IActivityManager mgr = ActivityManagerNative.getDefault();
6. **try** {
7. **mgr.attachApplication(mAppThread);**
8. } **catch** (RemoteException ex) {
9. }
10. } **else** {
11. }
12. }
13. **public** **static** **final** **void** main(String[] args) {
14. ActivityThread thread = **new** ActivityThread();
15. thread.attach(**false**);
16. Looper.loop();
17. thread.detach();
18. }
19. }

这个函数在进程中创建一个ActivityThread实例，然后调用它的attach函数，接着就进入消息循环了，直到最后进程退出。

函数attach最终调用了AtMS的远程接口ActivityManagerProxy的attachApplication函数，传入的参数是mAppThread，这是一个ApplicationThread类型的Binder对象，它的作用是用来进行进程间通信的。

#### ActivityThread. Attach

Asf

传进来的参数system为false。成员变量mAppThread是一个ApplicationThread实例，我们在前面已经描述过这个实例的作用，它是用来辅助ActivityThread来执行一些操作

**private void** attach(**boolean** system) {  
 sCurrentActivityThread = **this**;  
 mSystemThread = system;  
 **if** (!system) {  
 ViewRootImpl.addFirstDrawHandler(**new** Runnable() {  
 @Override  
 **public void** run() {  
 ensureJitEnabled();  
 }  
 });  
 android.ddm.DdmHandleAppName.setAppName(**"<pre-initialized>"**,  
 UserHandle.myUserId());  
 RuntimeInit.setApplicationObject(mAppThread.asBinder());  
 **final** IActivityManager mgr = ActivityManager.getService();  
 **try** {  
 mgr.attachApplication(mAppThread);  
 } **catch** (RemoteException ex) {  
 **throw** ex.rethrowFromSystemServer();  
 }  
 *// Watch for getting close to heap limit.* BinderInternal.addGcWatcher(**new** Runnable() {  
 @Override **public void** run() {  
 **if** (!mSomeActivitiesChanged) {  
 **return**;  
 }  
 Runtime runtime = Runtime.getRuntime();  
 **long** dalvikMax = runtime.maxMemory();  
 **long** dalvikUsed = runtime.totalMemory() - runtime.freeMemory();  
 **if** (dalvikUsed > ((3\*dalvikMax)/4)) {  
 **if** (DEBUG\_MEMORY\_TRIM) Slog.d(TAG, **"Dalvik max="** + (dalvikMax/1024)  
 + **" total="** + (runtime.totalMemory()/1024)  
 + **" used="** + (dalvikUsed/1024));  
 mSomeActivitiesChanged = **false**;

/\*\*

这里添加Gc的监听，如果超过虚拟机分配最大内存的 3/4，那么触发mgr.releaseSomeActivities

\*/

**try** {  
 mgr.releaseSomeActivities(mAppThread);  
 } **catch** (RemoteException e) {  
 **throw** e.rethrowFromSystemServer();  
 }  
 }  
 }  
 });  
 } **else** {  
 *// Don't set application object here -- if the system crashes,  
 // we can't display an alert, we just want to die die die.* android.ddm.DdmHandleAppName.setAppName(**"system\_process"**,  
 UserHandle.myUserId());  
 **try** {  
 mInstrumentation = **new** Instrumentation();  
 ContextImpl context = ContextImpl.createAppContext(  
 **this**, getSystemContext().mPackageInfo);  
 mInitialApplication = context.mPackageInfo.makeApplication(**true**, **null**);  
 mInitialApplication.onCreate();  
 } **catch** (Exception e) {  
 **throw new** RuntimeException(  
 **"Unable to instantiate Application():"** + e.toString(), e);  
 }  
 }  
  
 *// add dropbox logging to libcore* DropBox.setReporter(**new** DropBoxReporter());  
  
 ViewRootImpl.ConfigChangedCallback configChangedCallback  
 = (Configuration globalConfig) -> {  
 **synchronized** (mResourcesManager) {  
 *// We need to apply this change to the resources immediately, because upon returning  
 // the view hierarchy will be informed about it.* **if** (mResourcesManager.applyConfigurationToResourcesLocked(globalConfig,  
 **null** */\* compat \*/*)) {  
 updateLocaleListFromAppContext(mInitialApplication.getApplicationContext(),  
 mResourcesManager.getConfiguration().getLocales());  
  
 *// This actually changed the resources! Tell everyone about it.* **if** (mPendingConfiguration == **null** || mPendingConfiguration.isOtherSeqNewer(globalConfig)) {  
 mPendingConfiguration = globalConfig;  
 sendMessage(H.CONFIGURATION\_CHANGED, globalConfig);  
 }  
 }  
 }  
 };  
 ViewRootImpl.addConfigCallback(configChangedCallback);  
}

 我们很明显的看到了一个 BinderInternal.addGcWatcher()这个方法，我们可以看字面意思就知道，这个是一个Gc 回收的监听器，作用是在AppThread 作用域下的GcRoots Gc开始的时候，我们去监听这个GC回收，具体如何做到呢

### Step 25. ActivityManagerProxy.attachApplication

这个函数定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **void** attachApplication(IApplicationThread app) **throws** RemoteException
4. {
5. Parcel data = Parcel.obtain();
6. Parcel reply = Parcel.obtain();
7. data.writeInterfaceToken(IActivityManager.descriptor);
8. data.writeStrongBinder(app.asBinder());
9. mRemote.transact(ATTACH\_APPLICATION\_TRANSACTION, data, reply, 0);
10. reply.readException();
11. data.recycle();
12. reply.recycle();
13. }
14. }

这里通过Binder驱动程序，最后进入AtMS的attachApplication函数中。

## AMS

### Step 26. AtMS.attachApplication

1. **public** **final** **void** attachApplication(IApplicationThread thread) {
2. **synchronized** (**this**) {
3. **int** callingPid = Binder.getCallingPid();
4. **final** **long** origId = Binder.clearCallingIdentity();
5. attachApplicationLocked(thread, callingPid);
6. Binder.restoreCallingIdentity(origId);
7. }
8. }
9. }

这里将操作转发给attachApplicationLocked函数。

### Step 27. AtMS.attachApplicationLocked

1. **private** **final** **boolean** attachApplicationLocked(IApplicationThread thread,
2. **int** pid) {
3. // Find the application record that is being attached...  either via
4. // the pid if we are running in multiple processes, or just pull the
5. // next app record if we are emulating process with anonymous threads.
6. String processName = app.processName;
7. **try** {
8. thread.asBinder().linkToDeath(**new** AppDeathRecipient(
9. app, pid, thread), 0);
10. } **catch** (RemoteException e) {
11. ......
12. **return** **false**;
13. }
14. app.thread = thread;
15. app.curAdj = app.setAdj = -100;
16. app.curSchedGroup = Process.THREAD\_GROUP\_DEFAULT;
17. app.setSchedGroup = Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;
18. app.forcingToForeground = **null**;
19. app.foregroundServices = **false**;
20. app.debugging = **false**;
22. .....
24. thread.bindApplication(processName, appInfo, providers,
25. app.instr.mClass,
26. profilerInfo, app.instr.mArguments,
27. app.instr.mWatcher,
28. app.instr.mUiAutomationConnection, testMode,
29. mBinderTransactionTrackingEnabled, enableTrackAllocation,
30. isRestrictedBackupMode || !normalMode, app.persistent,
31. new Configuration(getGlobalConfiguration()), app.compat,
32. getCommonServicesLocked(app.isolated),
33. mCoreSettingsObserver.getCoreSettingsLocked(),
34. buildSerial);    ......
35. // See if the top visible activity is waiting to run in this process...
36. if (mStackSupervisor.attachApplicationLocked(app)) {
37. didSomething = true;
38. }
39. }

在前面的Step 23中，已经创建了一个ProcessRecord，这里首先通过pid将它取回来，放在app变量中，然后对app的其它成员进行初始化，最后调用mMainStack.realStartActivityLocked执行真正的Activity启动操作。这里要启动的Activity通过调用mMainStack.topRunningActivityLocked(null)从堆栈顶端取回来，这时候在堆栈顶端的Activity就是MainActivity了。

#### ApplicationThread .attachApplicationLocked

**ActivityThread.java$$ApplicationThread#bindApplication()**

**if** (services != **null**) {  
 *// Setup the service cache in the ServiceManager* ServiceManager.initServiceCache(services);  
}  
  
setCoreSettings(coreSettings);  
  
AppBindData data = **new** AppBindData();  
data.**processName** = processName;  
data.**appInfo** = appInfo;  
data.**providers** = providers;  
data.**instrumentationName** = instrumentationName;  
data.**instrumentationArgs** = instrumentationArgs;  
data.**instrumentationWatcher** = instrumentationWatcher;  
data.**instrumentationUiAutomationConnection** = instrumentationUiConnection;  
data.**debugMode** = debugMode;  
data.**enableBinderTracking** = enableBinderTracking;  
data.**trackAllocation** = trackAllocation;  
data.**restrictedBackupMode** = isRestrictedBackupMode;  
data.**persistent** = persistent;  
data.**config** = config;  
data.**compatInfo** = compatInfo;  
data.**initProfilerInfo** = profilerInfo;  
data.**buildSerial** = buildSerial;  
sendMessage(H.***BIND\_APPLICATION***, data);

#### sendMessage(H.BIND\_APPLICATION, data);

#### handleBindApplication(data);

### ActivityStackSupervisor .attachApplicationLocked

**if** (realStartActivityLocked(hr, app, **true**, **true**)) {  
 didSomething = **true**;  
}

### Step 28. ActivityStackSupervisor.realStartActivityLocked

1. **public** **class** ActivityStack {
3. ......
5. **final** **boolean** realStartActivityLocked(ActivityRecord r,
6. ProcessRecord app, **boolean** andResume, **boolean** checkConfig)
7. **throws** RemoteException {
9. ......
11. r.app = app;
13. ......
15. **int** idx = app.activities.indexOf(r);
16. **if** (idx < 0) {
17. app.activities.add(r);
18. }
20. ......
22. **try** {
23. ......
25. List<ResultInfo> results = **null**;
26. List<Intent> newIntents = **null**;
27. **if** (andResume) {
28. results = r.results;
29. newIntents = r.newIntents;
30. }
32. ......
34. app.thread.scheduleLaunchActivity(**new** Intent(r.intent), r,
35. System.identityHashCode(r),
36. r.info, r.icicle, results, newIntents, !andResume,
37. mService.isNextTransitionForward());
39. ......
41. } **catch** (RemoteException e) {
42. ......
43. }
45. ......
47. **return** **true**;
48. }
50. ......
52. }

这里最终通过app.thread进入到ApplicationThreadProxy的scheduleLaunchActivity函数中，注意，这里的第二个参数r，是一个ActivityRecord类型的Binder对象，用来作来这个Activity的token值。

当框架通过ApplicationThread的代理回调到ActivityThread的时候，将对应的步骤一种生成的token代理传入。 ActivityStackSupervisor.realStartActivityLocked()

### Step 29. ApplicationThreadProxy.scheduleLaunchActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
2. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
3. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
4. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward)
5. **throws** RemoteException {
6. Parcel data = Parcel.obtain();
7. data.writeInterfaceToken(IApplicationThread.descriptor);
8. intent.writeToParcel(data, 0);
9. data.writeStrongBinder(token);
10. data.writeInt(ident);
11. info.writeToParcel(data, 0);
12. data.writeBundle(state);
13. data.writeTypedList(pendingResults);
14. data.writeTypedList(pendingNewIntents);
15. data.writeInt(notResumed ? 1 : 0);
16. data.writeInt(isForward ? 1 : 0);
17. mRemote.transact(SCHEDULE\_LAUNCH\_ACTIVITY\_TRANSACTION, data, **null**,
18. IBinder.FLAG\_ONEWAY);
19. data.recycle();
20. }
21. }

这个函数最终通过Binder驱动程序进入到ApplicationThread的scheduleLaunchActivity函数中。

### Step 30. ApplicationThread.scheduleLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // we use token to identify this activity without having to send the
10. // activity itself back to the activity manager. (matters more with ipc)
11. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
12. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
13. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward) {
14. ActivityClientRecord r = **new** ActivityClientRecord();
16. r.token = token;
17. r.ident = ident;
18. r.intent = intent;
19. r.activityInfo = info;
20. r.state = state;
22. r.pendingResults = pendingResults;
23. r.pendingIntents = pendingNewIntents;
25. r.startsNotResumed = notResumed;
26. r.isForward = isForward;
28. queueOrSendMessage(H.LAUNCH\_ACTIVITY, r);
29. }
31. ......
33. }
35. ......
36. }

函数首先创建一个ActivityClientRecord实例，并且初始化它的成员变量，然后调用ActivityThread类的queueOrSendMessage函数进一步处理。

### Step 31. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // if the thread hasn't started yet, we don't have the handler, so just
10. // save the messages until we're ready.
11. **private** **final** **void** queueOrSendMessage(**int** what, Object obj) {
12. queueOrSendMessage(what, obj, 0, 0);
13. }
15. ......
17. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
18. **synchronized** (**this**) {
19. ......
20. Message msg = Message.obtain();
21. msg.what = what;
22. msg.obj = obj;
23. msg.arg1 = arg1;
24. msg.arg2 = arg2;
25. mH.sendMessage(msg);
26. }
27. }
29. ......
31. }
33. ......
34. }

函数把消息内容放在msg中，然后通过mH把消息分发出去，这里的成员变量mH我们在前面已经见过，消息分发出去后，最后会调用H类的handleMessage函数。

### Step 32. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** H **extends** Handler {
7. ......
9. **public** **void** handleMessage(Message msg) {
10. ......
11. **switch** (msg.what) {
12. **case** LAUNCH\_ACTIVITY: {
13. ActivityClientRecord r = (ActivityClientRecord)msg.obj;
15. r.packageInfo = getPackageInfoNoCheck(
16. r.activityInfo.applicationInfo);
17. handleLaunchActivity(r, **null**);
18. } **break**;
19. ......
20. }
22. ......
24. }
26. ......
27. }

这里最后调用ActivityThread类的handleLaunchActivity函数进一步处理。

### Step 33. ActivityThread.handleLaunchActivity

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {

        Activity a = performLaunchActivity(r, customIntent);  
  
**if** (a != **null**) {  
 r.**createdConfig** = **new** Configuration(**mConfiguration**);  
 reportSizeConfigurations(r);  
 Bundle oldState = r.**state**;  
 handleResumeActivity(r.**token**, **false**, r.**isForward**,  
 !r.**activity**.mFinished && !r.**startsNotResumed**, r.**lastProcessedSeq**, reason);

1. if (r.window == null && !a.mFinished && willBeVisible) {
2. //获得为当前Activity创建的窗口PhoneWindow对象
3. r.window = r.activity.getWindow();
4. //获取为窗口创建的视图DecorView对象
5. View decor = r.window.getDecorView();
6. decor.setVisibility(View.INVISIBLE);
7. //在attach函数中就为当前Activity创建了WindowManager对象
8. ViewManager wm = a.getWindowManager();
9. //得到该视图对象的布局参数
10. WindowManager.LayoutParams l = r.window.getAttributes();
11. //将视图对象保存到Activity的成员变量mDecor中
12. a.mDecor = decor;
13. l.type = WindowManager.LayoutParams.TYPE\_BASE\_APPLICATION;
14. l.softInputMode |= forwardBit;
15. ......
16. if (a.mVisibleFromClient && !a.mWindowAdded) {
17. a.mWindowAdded = true;
18. //将创建的视图对象DecorView添加到Activity的窗口管理器中
19. wm.addView(decor, l);
20. }
21. ......
22. if (r.activity.mVisibleFromClient) {
23. r.activity.makeVisible();
24. }
25. }
26. **if** (a != **null**) {
27. r.createdConfig = **new** Configuration(mConfiguration);
28. Bundle oldState = r.state;
29. handleResumeActivity(r.token, **false**, r.isForward);
30. }
31. }
32. }

这里首先调用performLaunchActivity函数来加载这个Activity类，即shy.luo.activity.MainActivity，然后调用它的onCreate函数，最后回到handleLaunchActivity函数时，

再调用handleResumeActivity函数来使这个Activity进入Resumed状态，即会调用这个Activity的onResume函数，这是遵循Activity的生命周期的。会调用Activity.makeVisible(),该方法继续调用便会执行到WindowManagerImpl.addView(), 该方法内部再调用WindowManagerGlobal.addView(),

### Step 34. ActivityThread.performLaunchActivity

1. **private** **final** Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
3. ActivityInfo aInfo = r.activityInfo;
4. **if** (r.packageInfo == **null**) {
5. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
6. Context.CONTEXT\_INCLUDE\_CODE);
7. }
9. ComponentName component = r.intent.getComponent();
10. **if** (component == **null**) {
11. component = r.intent.resolveActivity(
12. mInitialApplication.getPackageManager());
13. r.intent.setComponent(component);
14. }
16. **if** (r.activityInfo.targetActivity != **null**) {
17. component = **new** ComponentName(r.activityInfo.packageName,
18. r.activityInfo.targetActivity);
19. }
21. Activity activity = **null**;
22. **try** {
23. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
24. activity = mInstrumentation.newActivity(
25. cl, component.getClassName(), r.intent);
26. r.intent.setExtrasClassLoader(cl);

29. **try** {
30. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);
32. r.configCallback);
33. **if** (activity != **null**) {
34. ContextImpl appContext = **new** ContextImpl();
35. appContext.init(r.packageInfo, r.token, **this**);
36. appContext.setOuterContext(activity);
37. CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
38. Configuration config = **new** Configuration(mConfiguration);
39. ......
40. activity.attach(appContext, **this**, getInstrumentation(), r.token,
41. r.ident, app, r.intent, r.activityInfo, title, r.parent,
42. r.embeddedID, r.lastNonConfigurationInstance,
43. r.lastNonConfigurationChildInstances, config);
45. **if** (customIntent != **null**) {
46. activity.mIntent = customIntent;
47. }
48. r.lastNonConfigurationInstance = **null**;
49. r.lastNonConfigurationChildInstances = **null**;
50. activity.mStartedActivity = **false**;
51. **int** theme = r.activityInfo.getThemeResource();
52. **if** (theme != 0) {
53. activity.setTheme(theme);
54. }
56. activity.mCalled = **false**;
57. mInstrumentation.callActivityOnCreate(activity, r.state);
58. ......
59. r.activity = activity;
60. r.stopped = **true**;
61. **if** (!r.activity.mFinished) {
62. activity.performStart();
63. r.stopped = **false**;
64. }
65. **if** (!r.activity.mFinished) {
66. **if** (r.state != **null**) {
67. mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
68. }
69. }
70. **if** (!r.activity.mFinished) {
71. activity.mCalled = **false**;
72. mInstrumentation.callActivityOnPostCreate(activity, r.state);
73. **if** (!activity.mCalled) {
74. **throw** **new** SuperNotCalledException(
75. "Activity " + r.intent.getComponent().toShortString() +
76. " did not call through to super.onPostCreate()");
77. }
78. }
79. }
80. r.paused = **true**;
82. mActivities.put(r.token, r);
84. } **catch** (SuperNotCalledException e) {
85. ......
87. } **catch** (Exception e) {
88. ......
89. }
91. **return** activity;
92. }
94. ......
95. }

函数前面是收集要启动的Activity的相关信息，主要package和component信息：

1. ActivityInfo aInfo = r.activityInfo;
2. **if** (r.packageInfo == **null**) {
3. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
4. Context.CONTEXT\_INCLUDE\_CODE);
5. }
7. ComponentName component = r.intent.getComponent();
8. **if** (component == **null**) {
9. component = r.intent.resolveActivity(
10. mInitialApplication.getPackageManager());
11. r.intent.setComponent(component);
12. }
14. **if** (r.activityInfo.targetActivity != **null**) {
15. component = **new** ComponentName(r.activityInfo.packageName,
16. r.

#### ClassLoader

将shy.luo.activity.MainActivity类加载进来：

1. Activity activity = **null**;
2. **try** {
3. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
4. activity = mInstrumentation.newActivity(
5. cl, component.getClassName(), r.intent);
6. r.intent.setExtrasClassLoader(cl);
7. **if** (r.state != **null**) {
8. r.state.setClassLoader(cl);
9. }
10. } **catch** (Exception e) {
11. ......
12. }

#### 创建Application对象

这是根据AndroidManifest.xml配置文件中的Application标签的信息来创建的：

1. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);

#### 创建Activity的上下文信息

Context appContext = createBaseContextForActivity(r, activity); CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());

#### 创建 Configuration

Fgh

Configuration config = new Configuration(mCompatConfiguration);

if (r.overrideConfig != null) {

config.updateFrom(r.overrideConfig);

}

if (DEBUG\_CONFIGURATION) Slog.v(TAG, "Launching activity "

+ r.activityInfo.name + " with config " + config);

Window window = null;

if (r.mPendingRemoveWindow != null && r.mPreserveWindow) {

window = r.mPendingRemoveWindow;

r.mPendingRemoveWindow = null;

r.mPendingRemoveWindowManager = null;

}

#### attach

调用acitvity的attach，将创建的这些对象关联到activity

后面的代码主要创建Activity的上下文信息，并通过attach方法将这些上下文信息设置到MainActivity中去：

1. activity.attach(appContext, **this**, getInstrumentation(), r.token,
2. r.ident, app, r.intent, r.activityInfo, title, r.parent,
3. r.embeddedID, r.lastNonConfigurationInstance,
4. r.lastNonConfigurationChildInstances, config);

最后还要调用MainActivity的onCreate函数：

Instrumentation.callActivityOnCreate(activity, r.state);

这里不是直接调用MainActivity的onCreate函数，而是通过mInstrumentation的callActivityOnCreate函数来间接调用，前面我们说过，mInstrumentation在这里的作用是监控Activity与系统的交互操作，相当于是系统运行日志。

#### Step 35. MainActivity.onCreate

1. **public** **class** MainActivity **extends** Activity  **implements** OnClickListener {
2. @Override
3. **public** **void** onCreate(Bundle savedInstanceState) {          Log.i(LOG\_TAG, "Main Activity Created.");
4. }
6. ......

Dfg

handleLaunchActivity()：会调用Activity.onCreate(), 该方法内再调用setContentView(),经过AMS与WMS的各种交互,层层调用后,进入step2

handleResumeActivity()：会调用Activity.makeVisible(),该方法继续调用便会执行到WindowManagerImpl.addView(), 该方法内部再调用WindowManagerGlobal.addView(),

if (app.persistent) {

Watchdog.getInstance().processStarted(app.processName

### ActivityThread .handleResumeActivity

unscheduleGcIdler();  
**mSomeActivitiesChanged** = **true**;  
  
*// TODO Push resumeArgs into the activity for consideration*r = performResumeActivity(token, clearHide, reason);

**if** (!r.**onlyLocalRequest**) {  
 r.**nextIdle** = **mNewActivities**;  
 **mNewActivities** = r;  
 **if** (***localLOGV***) Slog.v(  
 ***TAG***, **"Scheduling idle handler for "** + r);  
 Looper.*myQueue*().addIdleHandler(**new** Idler());  
}

*// Tell the activity manager we have resumed.***if** (reallyResume) {  
 **try** {  
 ActivityManager.*getService*().activityResumed(token);  
 } **catch** (RemoteException ex) {  
 **throw** ex.rethrowFromSystemServer();  
 }  
}

#### performResumeActivity

##### Activity. performResume

**final void** performResume(**boolean** followedByPause, String reason) {  
 performRestart(**true** */\* start \*/*, reason);

**mInstrumentation**.callActivityOnResume(**this**);

##### Instrumentation. callActivityOnResume

**public void** callActivityOnResume(Activity activity) {  
 activity.mResumed = **true**;  
 activity.onResume();

#### Looper.*myQueue*().addIdleHandler(new Idler());

## 实战问题

////

/\*

dji\_custom for 3rd app when call Application.OnCreate

\*/

mInstrumentation.callApplicationOnCreate(app);

DjiCoreHelper.getIns().callApplicationOnCreate(app);

public void callApplicationOnCreate(Application app){

if (app == null){

return;

}

String processName = app.getApplicationInfo().processName;

if(TextUtils.equals(processName, PK\_NAME\_BROWSER)){

SharedPreferences sp = app.getSharedPreferences("settings", Context.MODE\_PRIVATE);

if( sp.contains(djiInitKey) ){

LOGD("djiInited, return");

return;

}

Editor editor = sp.edit();

if ( isCn() ){

editor.putInt("search", 9);//baidu

}

editor.putString("home", "https://www.dji.com")//home page

.putInt("Theme", 1)//1 is better,2,amoled

.putString(djiInitKey, "1").commit();;

}/\*else if(TextUtils.equals(processName, PK\_NAME\_PINYIN)){

SharedPreferences sp = app.getSharedPreferences(processName, Context.MODE\_PRIVATE);

if( sp.contains(djiInitKey) ){

// return;

}

Editor editor = sp.edit();

editor.putString("china\_preload\_notice\_shown", "false").commit();

}\*/

}

## 小结

这样，MainActivity就启动起来了，整个应用程序也启动起来了。

       整个应用程序的启动过程要执行很多步骤，但是整体来看，主要分为以下五个阶段：

       一. Step1 - Step 11：Launcher通过Binder进程间通信机制通知AtMS，它要启动一个Activity；

       二. Step 12 - Step 16：AtMS通过Binder进程间通信机制通知Launcher进入Paused状态；

       三. Step 17 - Step 24：Launcher通过Binder进程间通信机制通知AtMS，它已经准备就绪进入Paused状态，于是AtMS就创建一个新的进程，用来启动一个ActivityThread实例，即将要启动的Activity就是在这个ActivityThread实例中运行；

       四. Step 25 - Step 27：ActivityThread通过Binder进程间通信机制将一个ApplicationThread类型的Binder对象传递给AtMS，以便以后AtMS能够通过这个Binder对象和它进行通信；

       五. Step 28 - Step 35：AtMS通过Binder进程间通信机制通知ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了。

       这里不少地方涉及到了Binder进程间通信机制，相关资料请参考[Android进程间通信（IPC）机制Binder简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6618363" \t "_blank)一文。

       这样，应用程序的启动过程就介绍完了，它实质上是启动应用程序的默认Activity，在下一篇文章中，我们将介绍在应用程序内部启动另一个Activity的过程，即新的Activity与启动它的Activity将会在同一个进程（Process）和任务（Task）运行，敬请关注。

1、AMS接到客户端的调用过后，开始启动，把启动的工作交给了ActivityStarter的startActivityMayWait

2、startActivityMayWait分了三步：

2.1>先从PMS中搜索匹配信息

2.2>然后startActivityUnchecked启动activity startActivityUnchecked：处理launchmode、以怎样的task来启动activity、打开动画处理、由resumeTopActivityInnerLocked分两个方向走

2.2.1>>如果mResumedActivity不为空，则需要先暂停这个Activity。由startPausingLocked来完成

2.2.2>>当mResumedActivity为空时，若待启动的Activity对应的应用存在，那么仅需要重新启动该Activity 否则，需要调用ActivityStackSupervisor的startSpecificActivityLocked函数，启动整个进程。

2.2.2.1>>>通知进程启动目标Activity，此流程最后调用到performLaunchActivity直接创建了activity

2.2.2.2>>>如果进程不存在需要创建进程来运行

2.3>最后处理返回值。

## 参考

[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)

[**Android深入四大组件（一）应用程序启动过程**](http://blog.csdn.net/itachi85/article/details/69388942)

<http://blog.csdn.net/itachi85/article/details/69388942>

startService启动过程分析

<http://gityuan.com/2016/03/06/start-service/>

深入理解ActivityManagerService

<http://wiki.jikexueyuan.com/project/deep-android-v2/activity.html>

# 参考

[Android系统应用的开发和测试](http://www.epubit.com.cn/book/onlinechapter/30644)

[Android 源代码目录结构1 - bionic](http://blog.csdn.net/kickxxx/article/details/6927272)

[Android init.rc文件解析过程详解(一)](http://blog.csdn.net/mk1111/article/details/16357327)

[Android的权限机制之—— “沙箱”机制sharedUserId和签名](http://dengzhangtao.iteye.com/blog/1989065)

Android系统开篇：

<http://gityuan.com/android/#%E4%B8%80%E5%BC%95%E8%A8%80>

**Yuanhuihui**

<https://github.com/yuanhuihui?tab=repositories>